Spring Boot Reference Documentation

Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons, Vedran Pavić, Jay Bryant, Madhura Bhave, Eddú Meléndez, Scott Frederick, Moritz Halbritter

Version 3.1.6

Table of Contents

1. Legal . . . 2
2. Getting Help . . . 3
3. Documentation Overview . . . 4
    3.1. First Steps . . . 4
    3.2. Upgrading From an Earlier Version . . . 4
    3.3. Developing With Spring Boot . . . 4
    3.4. Learning About Spring Boot Features . . . 4
    3.5. Web . . . 5
    3.6. Data . . . 5
    3.7. Messaging . . . 5
    3.8. IO . . . 5
    3.9. Container Images . . . 6
    3.10. Moving to Production . . . 6
    3.11. GraalVM Native Images . . . 6
    3.12. Advanced Topics . . . 6
4. Getting Started . . . 7
    4.1. Introducing Spring Boot . . . 7
    4.2. System Requirements . . . 7
        4.2.1. Servlet Containers . . . 7
        4.2.2. GraalVM Native Images . . . 8
    4.3. Installing Spring Boot . . . 8
        4.3.1. Installation Instructions for the Java Developer . . . 8
            Maven Installation . . . 8
            Gradle Installation . . . 9
        4.3.2. Installing the Spring Boot CLI . . . 9
            Manual Installation . . . 9
            Installation with SDKMAN! . . . 10
            OSX Homebrew Installation . . . 10
            MacPorts Installation . . . 11
            Command-line Completion . . . 11
            Windows Scoop Installation . . . 11
    4.4. Developing Your First Spring Boot Application . . . 12
        4.4.1. Prerequisites . . . 12
            Maven . . . 12
            Gradle . . . 12
        4.4.2. Setting up the project with Maven . . . 13
        4.4.3. Setting up the project with Gradle . . . 14
        4.4.4. Adding Classpath Dependencies . . . 14

Maven . . . 14
Gradle . . . 15
4.4.5. Writing the Code . . . 15
The @RestController and @RequestMapping Annotations . . . 17
The @SpringBootApplication Annotation . . . 17
The “main” Method . . . 17
4.4.6. Running the Example . . . 17
Maven . . . 17
Gradle . . . 18
4.4.7. Creating an Executable Jar . . . 19
Maven . . . 19
Gradle . . . 20
4.5. What to Read Next . . . 21
5. Upgrading Spring Boot . . . 22
5.1. Upgrading From 1.x . . . 22
5.2. Upgrading to a New Feature Release . . . 22
5.3. Upgrading the Spring Boot CLI . . . 22
5.4. What to Read Next . . . 22
6. Developing with Spring Boot . . . 24
6.1. Build Systems . . . 24
6.1.1. Dependency Management . . . 24
6.1.2. Maven . . . 24
6.1.3. Gradle . . . 24
6.1.4. Ant . . . 25
6.1.5. Starters . . . 26
6.2. Structuring Your Code . . . 30
6.2.1. Using the “default” Package . . . 30
6.2.2. Locating the Main Application Class . . . 30
6.3. Configuration Classes . . . 32
6.3.1. Importing Additional Configuration Classes . . . 32
6.3.2. Importing XML Configuration . . . 32
6.4. Auto-configuration . . . 32
6.4.1. Gradually Replacing Auto-configuration . . . 33
6.4.2. Disabling Specific Auto-configuration Classes . . . 33
6.4.3. Auto-configuration Packages . . . 34
6.5. Spring Beans and Dependency Injection . . . 34
6.6. Using the @SpringBootApplication Annotation . . . 36
6.7. Running Your Application . . . 39
6.7.1. Running From an IDE . . . 39
6.7.2. Running as a Packaged Application . . . 39
6.7.3. Using the Maven Plugin . . . 39

6.7.4. Using the Gradle Plugin . . . 40
6.7.5. Hot Swapping . . . 40
6.8. Developer Tools . . . 40
6.8.1. Diagnosing Classloading Issues . . . 41
6.8.2. Property Defaults . . . 41
6.8.3. Automatic Restart . . . 42
Logging Changes in Condition Evaluation . . . 44
Excluding Resources . . . 44
Watching Additional Paths . . . 45
Disabling Restart . . . 45
Using a Trigger File . . . 46
Customizing the Restart Classloader . . . 47
Known Limitations . . . 48
6.8.4. LiveReload . . . 48
6.8.5. Global Settings . . . 48
Configuring File System Watcher . . . 49
6.8.6. Remote Applications . . . 50
Running the Remote Client Application . . . 50
Remote Update . . . 51
6.9. Packaging Your Application for Production . . . 52
6.10. What to Read Next . . . 52
7. Core Features . . . 53
7.1. SpringApplication . . . 53
7.1.1. Startup Failure . . . 54
7.1.2. Lazy Initialization . . . 55
7.1.3. Customizing the Banner . . . 56
7.1.4. Customizing SpringApplication . . . 57
7.1.5. Fluent Builder API . . . 58
7.1.6. Application Availability . . . 59
Liveness State . . . 59
Readiness State . . . 59
Managing the Application Availability State . . . 59
7.1.7. Application Events and Listeners . . . 63
7.1.8. Web Environment . . . 64
7.1.9. Accessing Application Arguments . . . 64
7.1.10. Using the ApplicationRunner or CommandLineRunner . . . 65
7.1.11. Application Exit . . . 66
7.1.12. Admin Features . . . 68
7.1.13. Application Startup tracking . . . 68
7.2. Externalized Configuration . . . 69
7.2.1. Accessing Command Line Properties . . . 71

7.2.2. JSON Application Properties . . . 72
7.2.3. External Application Properties . . . 72
Optional Locations . . . 74
Wildcard Locations . . . 74
Profile Specific Files . . . 75
Importing Additional Data . . . 76
Importing Extensionless Files . . . 78
Using Configuration Trees . . . 78
Property Placeholders . . . 81
Working With Multi-Document Files . . . 81
Activation Properties . . . 82
7.2.4. Encrypting Properties . . . 83
7.2.5. Working With YAML . . . 83
Mapping YAML to Properties . . . 83
Directly Loading YAML . . . 84
7.2.6. Configuring Random Values . . . 85
7.2.7. Configuring System Environment Properties . . . 85
7.2.8. Type-safe Configuration Properties . . . 85
JavaBean Properties Binding . . . 86
Constructor Binding . . . 89
Enabling @ConfigurationProperties-annotated Types . . . 92
Using @ConfigurationProperties-annotated Types . . . 94
Third-party Configuration . . . 96
Relaxed Binding . . . 97
Merging Complex Types . . . 100
Properties Conversion . . . 104
@ConfigurationProperties Validation . . . 111
@ConfigurationProperties vs. @Value . . . 114
7.3. Profiles . . . 115
7.3.1. Adding Active Profiles . . . 117
7.3.2. Profile Groups . . . 118
7.3.3. Programmatically Setting Profiles . . . 119
7.3.4. Profile-specific Configuration Files . . . 119
7.4. Logging . . . 119
7.4.1. Log Format . . . 119
7.4.2. Console Output . . . 120
Color-coded Output . . . 121
7.4.3. File Output . . . 122
7.4.4. File Rotation . . . 122
7.4.5. Log Levels . . . 123
7.4.6. Log Groups . . . 123

7.4.7. Using a Log Shutdown Hook . . . 124
7.4.8. Custom Log Configuration . . . 125
7.4.9. Logback Extensions . . . 127
Profile-specific Configuration . . . 128
Environment Properties . . . 128
7.4.10. Log4j2 Extensions . . . 128
Profile-specific Configuration . . . 129
Environment Properties Lookup . . . 129
Log4j2 System Properties . . . 130
7.5. Internationalization . . . 130
7.6. Aspect-Oriented Programming . . . 131
7.7. JSON . . . 131
7.7.1. Jackson . . . 131
Custom Serializers and Deserializers . . . 131
Mixins . . . 135
7.7.2. Gson . . . 135
7.7.3. JSON-B . . . 136
7.8. Task Execution and Scheduling . . . 136
7.9. Testing . . . 137
7.9.1. Test Scope Dependencies . . . 138
7.9.2. Testing Spring Applications . . . 138
7.9.3. Testing Spring Boot Applications . . . 138
Detecting Web Application Type . . . 139
Detecting Test Configuration . . . 140
Using the Test Configuration Main Method . . . 141
Excluding Test Configuration . . . 143
Using Application Arguments . . . 144
Testing With a Mock Environment . . . 145
Testing With a Running Server . . . 149
Customizing WebTestClient . . . 151
Using JMX . . . 152
Using Metrics . . . 153
Using Tracing . . . 153
Mocking and Spying Beans . . . 153
Auto-configured Tests . . . 156
Auto-configured JSON Tests . . . 157
Auto-configured Spring MVC Tests . . . 160
Auto-configured Spring WebFlux Tests . . . 164
Auto-configured Spring GraphQL Tests . . . 167
Auto-configured Data Cassandra Tests . . . 172
Auto-configured Data Couchbase Tests . . . 173

Auto-configured Data Elasticsearch Tests . . . 174
Auto-configured Data JPA Tests . . . 175
Auto-configured JDBC Tests . . . 178
Auto-configured Data JDBC Tests . . . 179
Auto-configured Data R2DBC Tests . . . 179
Auto-configured jOOQ Tests . . . 180
Auto-configured Data MongoDB Tests . . . 181
Auto-configured Data Neo4j Tests . . . 182
Auto-configured Data Redis Tests . . . 183
Auto-configured Data LDAP Tests . . . 184
Auto-configured REST Clients . . . 186
Auto-configured Spring REST Docs Tests . . . 188
Auto-configured Spring Web Services Tests . . . 199
Additional Auto-configuration and Slicing . . . 203
User Configuration and Slicing . . . 204
Using Spock to Test Spring Boot Applications . . . 208
7.9.4. Testcontainers . . . 208
Service Connections . . . 210
Dynamic Properties . . . 213
7.9.5. Test Utilities . . . 215
ConfigDataApplicationContextInitializer . . . 215
TestPropertyValues . . . 216
OutputCapture . . . 217
TestRestTemplate . . . 218
7.10. Docker Compose Support . . . 222
7.10.1. Service Connections . . . 223
7.10.2. Custom Images . . . 224
7.10.3. Skipping Specific Containers . . . 224
7.10.4. Using a Specific Compose File . . . 225
7.10.5. Waiting for Container Readiness . . . 225
7.10.6. Controlling the Docker Compose Lifecycle . . . 226
7.10.7. Activating Docker Compose Profiles . . . 227
7.11. Testcontainers Support . . . 227
7.11.1. Using Testcontainers at Development Time . . . 227
Contributing Dynamic Properties at Development Time . . . 230
Importing Testcontainer Declaration Classes . . . 231
Using DevTools with Testcontainers at Development Time . . . 233
7.12. Creating Your Own Auto-configuration . . . 234
7.12.1. Understanding Auto-configured Beans . . . 234
7.12.2. Locating Auto-configuration Candidates . . . 235
7.12.3. Condition Annotations . . . 235

Class Conditions . . . 236
Bean Conditions . . . 237
Property Conditions . . . 239
Resource Conditions . . . 239
Web Application Conditions . . . 239
SpEL Expression Conditions . . . 239
7.12.4. Testing your Auto-configuration . . . 239
Simulating a Web Context . . . 242
Overriding the Classpath . . . 243
7.12.5. Creating Your Own Starter . . . 243
Naming . . . 244
Configuration keys . . . 244
The “autoconfigure” Module . . . 246
Starter Module . . . 247
7.13. Kotlin Support . . . 248
7.13.1. Requirements . . . 248
7.13.2. Null-safety . . . 248
7.13.3. Kotlin API . . . 249
runApplication . . . 249
Extensions . . . 249
7.13.4. Dependency management . . . 250
7.13.5. @ConfigurationProperties . . . 250
7.13.6. Testing . . . 250
7.13.7. Resources . . . 251
Further reading . . . 251
Examples . . . 251
7.14. SSL . . . 251
7.14.1. Configuring SSL With Java KeyStore Files . . . 251
7.14.2. Configuring SSL With PEM-encoded Certificates . . . 252
7.14.3. Applying SSL Bundles . . . 254
7.14.4. Using SSL Bundles . . . 255
7.15. What to Read Next . . . 256
8. Web . . . 257
8.1. Servlet Web Applications . . . 257
8.1.1. The “Spring Web MVC Framework” . . . 257
Spring MVC Auto-configuration . . . 263
Spring MVC Conversion Service . . . 263
HttpMessageConverters . . . 264
MessageCodesResolver . . . 265
Static Content . . . 265
Welcome Page . . . 268

Custom Favicon . . . 268
Path Matching and Content Negotiation . . . 268
ConfigurableWebBindingInitializer . . . 270
Template Engines . . . 270
Error Handling . . . 270
CORS Support . . . 280
8.1.2. JAX-RS and Jersey . . . 281
8.1.3. Embedded Servlet Container Support . . . 283
Servlets, Filters, and Listeners . . . 283
Servlet Context Initialization . . . 283
The ServletWebServerApplicationContext . . . 284
Customizing Embedded Servlet Containers . . . 285
JSP Limitations . . . 290
8.2. Reactive Web Applications . . . 290
8.2.1. The “Spring WebFlux Framework” . . . 290
Spring WebFlux Auto-configuration . . . 296
Spring WebFlux Conversion Service . . . 296
HTTP Codecs with HttpMessageReaders and HttpMessageWriters . . . 297
Static Content . . . 298
Welcome Page . . . 299
Template Engines . . . 299
Error Handling . . . 299
Web Filters . . . 305
8.2.2. Embedded Reactive Server Support . . . 305
Customizing Reactive Servers . . . 305
8.2.3. Reactive Server Resources Configuration . . . 308
8.3. Graceful Shutdown . . . 308
8.4. Spring Security . . . 309
8.4.1. MVC Security . . . 310
8.4.2. WebFlux Security . . . 310
8.4.3. OAuth2 . . . 312
Client . . . 312
Resource Server . . . 318
Authorization Server . . . 320
8.4.4. SAML 2.0 . . . 323
Relying Party . . . 323
8.5. Spring Session . . . 326
8.6. Spring for GraphQL . . . 327
8.6.1. GraphQL Schema . . . 328
8.6.2. GraphQL RuntimeWiring . . . 329
8.6.3. Querydsl and QueryByExample Repositories Support . . . 330

8.6.4. Transports . . . 331
HTTP and WebSocket . . . 331
RSocket . . . 331
8.6.5. Exception Handling . . . 333
8.6.6. GraphiQL and Schema printer . . . 333
8.7. Spring HATEOAS . . . 333
8.8. What to Read Next . . . 334
9. Data . . . 335
9.1. SQL Databases . . . 335
9.1.1. Configure a DataSource . . . 335
Embedded Database Support . . . 335
Connection to a Production Database . . . 336
DataSource Configuration . . . 336
Supported Connection Pools . . . 337
Connection to a JNDI DataSource . . . 338
9.1.2. Using JdbcTemplate . . . 338
9.1.3. JPA and Spring Data JPA . . . 340
Entity Classes . . . 340
Spring Data JPA Repositories . . . 342
Spring Data Envers Repositories . . . 344
Creating and Dropping JPA Databases . . . 344
Open EntityManager in View . . . 345
9.1.4. Spring Data JDBC . . . 345
9.1.5. Using H2's Web Console . . . 346
Changing the H2 Console's Path . . . 346
Accessing the H2 Console in a Secured Application . . . 346
9.1.6. Using jOOQ . . . 348
Code Generation . . . 348
Using DSLContext . . . 349
jOOQ SQL Dialect . . . 351
Customizing jOOQ . . . 351
9.1.7. Using R2DBC . . . 351
Embedded Database Support . . . 354
Using DatabaseClient . . . 355
Spring Data R2DBC Repositories . . . 355
9.2. Working with NoSQL Technologies . . . 356
9.2.1. Redis . . . 357
Connecting to Redis . . . 357
9.2.2. MongoDB . . . 359
Connecting to a MongoDB Database . . . 359
MongoTemplate . . . 363

Spring Data MongoDB Repositories . . . 364
9.2.3. Neo4j . . . 365
Connecting to a Neo4j Database . . . 365
Spring Data Neo4j Repositories . . . 367
9.2.4. Elasticsearch . . . 370
Connecting to Elasticsearch Using REST clients . . . 370
Connecting to Elasticsearch by Using Spring Data . . . 371
Spring Data Elasticsearch Repositories . . . 372
9.2.5. Cassandra . . . 373
Connecting to Cassandra . . . 373
Spring Data Cassandra Repositories . . . 376
9.2.6. Couchbase . . . 377
Connecting to Couchbase . . . 377
Spring Data Couchbase Repositories . . . 378
9.2.7. LDAP . . . 380
Connecting to an LDAP Server . . . 380
Spring Data LDAP Repositories . . . 381
Embedded In-memory LDAP Server . . . 382
9.2.8. InfluxDB . . . 383
Connecting to InfluxDB . . . 383
9.3. What to Read Next . . . 383
10. Messaging . . . 384
10.1. JMS . . . 384
10.1.1. ActiveMQ "Classic" Support . . . 384
10.1.2. ActiveMQ Artemis Support . . . 385
10.1.3. Using a JNDI ConnectionFactory . . . 387
10.1.4. Sending a Message . . . 387
10.1.5. Receiving a Message . . . 388
10.2. AMQP . . . 392
10.2.1. RabbitMQ Support . . . 392
10.2.2. Sending a Message . . . 393
10.2.3. Sending a Message To A Stream . . . 395
10.2.4. Receiving a Message . . . 396
10.3. Apache Kafka Support . . . 399
10.3.1. Sending a Message . . . 400
10.3.2. Receiving a Message . . . 401
10.3.3. Kafka Streams . . . 402
10.3.4. Additional Kafka Properties . . . 403
10.3.5. Testing with Embedded Kafka . . . 405
10.4. RSocket . . . 407
10.4.1. RSocket Strategies Auto-configuration . . . 407

10.4.2. RSocket server Auto-configuration . . . 407
10.4.3. Spring Messaging RSocket support . . . 408
10.4.4. Calling RSocket Services with RSocketRequester . . . 408
10.5. Spring Integration . . . 410
10.6. WebSockets . . . 412
10.7. What to Read Next . . . 412
11. IO . . . 413
11.1. Caching . . . 413
11.1.1. Supported Cache Providers . . . 414
Generic . . . 416
JCache (JSR-107) . . . 416
Hazelcast . . . 417
Infinispan . . . 417
Couchbase . . . 418
Redis . . . 420
Caffeine . . . 422
Cache2k . . . 423
Simple . . . 424
None . . . 424
11.2. Hazelcast . . . 425
11.3. Quartz Scheduler . . . 426
11.4. Sending Email . . . 429
11.5. Validation . . . 430
11.6. Calling REST Services . . . 431
11.6.1. RestTemplate . . . 432
RestTemplate HTTP Client . . . 433
RestTemplate Customization . . . 433
RestTemplate SSL Support . . . 437
11.6.2. WebClient . . . 438
WebClient Runtime . . . 439
WebClient Customization . . . 440
WebClient SSL Support . . . 440
11.7. Web Services . . . 442
11.7.1. Calling Web Services with WebServiceTemplate . . . 443
11.8. Distributed Transactions With JTA . . . 446
11.8.1. Using a Jakarta EE Managed Transaction Manager . . . 446
11.8.2. Mixing XA and Non-XA JMS Connections . . . 446
11.8.3. Supporting an Embedded Transaction Manager . . . 447
11.9. What to Read Next . . . 447
12. Container Images . . . 448
12.1. Efficient Container Images . . . 448

12.1.1. Layering Docker Images . . . 448
12.2. Dockerfiles . . . 449
12.3. Cloud Native Buildpacks . . . 450
12.4. What to Read Next . . . 451
13. Production-ready Features . . . 452
13.1. Enabling Production-ready Features . . . 452
13.2. Endpoints . . . 452
13.2.1. Enabling Endpoints . . . 454
13.2.2. Exposing Endpoints . . . 455
13.2.3. Security . . . 456
Cross Site Request Forgery Protection . . . 459
13.2.4. Configuring Endpoints . . . 459
13.2.5. Sanitize Sensitive Values . . . 460
13.2.6. Hypermedia for Actuator Web Endpoints . . . 461
13.2.7. CORS Support . . . 461
13.2.8. Implementing Custom Endpoints . . . 462
Receiving Input . . . 462
Custom Web Endpoints . . . 463
Servlet Endpoints . . . 465
Controller Endpoints . . . 465
13.2.9. Health Information . . . 465
Auto-configured HealthIndicators . . . 466
Writing Custom HealthIndicators . . . 467
Reactive Health Indicators . . . 470
Auto-configured ReactiveHealthIndicators . . . 472
Health Groups . . . 472
DataSource Health . . . 474
13.2.10. Kubernetes Probes . . . 474
Checking External State With Kubernetes Probes . . . 476
Application Lifecycle and Probe States . . . 477
13.2.11. Application Information . . . 477
Auto-configured InfoContributors . . . 478
Custom Application Information . . . 478
Git Commit Information . . . 479
Build Information . . . 480
Java Information . . . 480
OS Information . . . 480
Writing Custom InfoContributors . . . 480
13.3. Monitoring and Management Over HTTP . . . 481
13.3.1. Customizing the Management Endpoint Paths . . . 482
13.3.2. Customizing the Management Server Port . . . 483

13.3.3. Configuring Management-specific SSL . . . 483
13.3.4. Customizing the Management Server Address . . . 485
13.3.5. Disabling HTTP Endpoints . . . 485
13.4. Monitoring and Management over JMX . . . 486
13.4.1. Customizing MBean Names . . . 486
13.4.2. Disabling JMX Endpoints . . . 487
13.5. Observability . . . 487
13.6. Loggers . . . 488
13.6.1. Configure a Logger . . . 489
13.7. Metrics . . . 489
13.7.1. Getting started . . . 490
13.7.2. Supported Monitoring Systems . . . 493
AppOptics . . . 493
Atlas . . . 494
Datadog . . . 494
Dynatrace . . . 495
Elastic . . . 498
Ganglia . . . 499
Graphite . . . 499
Humio . . . 501
Influx . . . 502
JMX . . . 502
KairosDB . . . 504
New Relic . . . 504
OpenTelemetry . . . 506
Prometheus . . . 506
SignalFx . . . 507
Simple . . . 507
Stackdriver . . . 508
StatsD . . . 508
Wavefront . . . 509
13.7.3. Supported Metrics and Meters . . . 510
JVM Metrics . . . 510
System Metrics . . . 511
Application Startup Metrics . . . 511
Logger Metrics . . . 511
Task Execution and Scheduling Metrics . . . 511
Spring MVC Metrics . . . 511
Spring WebFlux Metrics . . . 512
Jersey Server Metrics . . . 512
HTTP Client Metrics . . . 513

Tomcat Metrics . . . 513
Cache Metrics . . . 513
Spring Batch Metrics . . . 514
Spring GraphQL Metrics . . . 514
DataSource Metrics . . . 514
Hibernate Metrics . . . 514
Spring Data Repository Metrics . . . 515
RabbitMQ Metrics . . . 515
Spring Integration Metrics . . . 515
Kafka Metrics . . . 516
MongoDB Metrics . . . 516
Jetty Metrics . . . 519
@Timed Annotation Support . . . 519
Redis Metrics . . . 519
13.7.4. Registering Custom Metrics . . . 519
13.7.5. Customizing Individual Metrics . . . 521
Common Tags . . . 522
Per-meter Properties . . . 523
13.7.6. Metrics Endpoint . . . 524
13.7.7. Integration with Micrometer Observation . . . 524
13.8. Tracing . . . 524
13.8.1. Supported Tracers . . . 524
13.8.2. Getting Started . . . 525
13.8.3. Propagating Traces . . . 526
13.8.4. Tracer Implementations . . . 526
OpenTelemetry With Zipkin . . . 527
OpenTelemetry With Wavefront . . . 527
OpenTelemetry With OTLP . . . 527
OpenZipkin Brave With Zipkin . . . 527
OpenZipkin Brave With Wavefront . . . 527
13.8.5. Integration with Micrometer Observation . . . 528
13.8.6. Creating Custom Spans . . . 528
13.8.7. Baggage . . . 528
13.8.8. Tests . . . 529
13.9. Auditing . . . 529
13.9.1. Custom Auditing . . . 530
13.10. Recording HTTP Exchanges . . . 530
13.10.1. Custom HTTP Exchange Recording . . . 530
13.11. Process Monitoring . . . 530
13.11.1. Extending Configuration . . . 531
13.11.2. Programmatically Enabling Process Monitoring . . . 531

13.12. Cloud Foundry Support . . . 531
13.12.1. Disabling Extended Cloud Foundry Actuator Support . . . 531
13.12.2. Cloud Foundry Self-signed Certificates . . . 532
13.12.3. Custom Context Path . . . 532
13.13. What to Read Next . . . 537
14. Deploying Spring Boot Applications . . . 538
14.1. Deploying to the Cloud . . . 538
14.1.1. Cloud Foundry . . . 538
Binding to Services . . . 540
14.1.2. Kubernetes . . . 541
Kubernetes Container Lifecycle . . . 541
14.1.3. Heroku . . . 542
14.1.4. OpenShift . . . 543
14.1.5. Amazon Web Services (AWS) . . . 544
AWS Elastic Beanstalk . . . 544
Summary . . . 545
14.1.6. CloudCaptain and Amazon Web Services . . . 545
14.1.7. Azure . . . 546
14.1.8. Google Cloud . . . 546
14.2. Installing Spring Boot Applications . . . 547
14.2.1. Installation as a systemd Service . . . 547
14.2.2. Installation as an init.d Service (System V) . . . 548
Securing an init.d Service . . . 550
Customizing the Startup Script . . . 551
14.2.3. Microsoft Windows Services . . . 554
14.3. Efficient deployments . . . 554
14.3.1. Unpacking the Executable JAR . . . 554
14.3.2. Using Ahead-of-time Processing With the JVM . . . 554
14.4. What to Read Next . . . 555
15. GraalVM Native Image Support . . . 556
15.1. Introducing GraalVM Native Images . . . 556
15.1.1. Key Differences with JVM Deployments . . . 556
15.1.2. Understanding Spring Ahead-of-Time Processing . . . 557
Source Code Generation . . . 557
Hint File Generation . . . 560
Proxy Class Generation . . . 560
15.2. Developing Your First GraalVM Native Application . . . 561
15.2.1. Sample Application . . . 561
15.2.2. Building a Native Image Using Buildpacks . . . 562
System Requirements . . . 562
Using Maven . . . 562

Using Gradle . . . 563
Running the example . . . 563
15.2.3. Building a Native Image using Native Build Tools . . . 564
Prerequisites . . . 564
Using Maven . . . 564
Using Gradle . . . 565
Running the Example . . . 565
15.3. Testing GraalVM Native Images . . . 566
15.3.1. Testing Ahead-of-time Processing With the JVM . . . 566
15.3.2. Testing With Native Build Tools . . . 567
Using Maven . . . 567
Using Gradle . . . 568
15.4. Advanced Native Images Topics . . . 568
15.4.1. Nested Configuration Properties . . . 568
15.4.2. Converting a Spring Boot Executable Jar . . . 570
Using Buildpacks . . . 570
Using GraalVM native-image . . . 571
15.4.3. Using the Tracing Agent . . . 572
Launch the Application Directly . . . 572
15.4.4. Custom Hints . . . 572
Testing custom hints . . . 573
15.4.5. Known Limitations . . . 574
15.5. What to Read Next . . . 574
16. Spring Boot CLI . . . 575
16.1. Installing the CLI . . . 575
16.2. Using the CLI . . . 575
16.2.1. Initialize a New Project . . . 577
16.2.2. Using the Embedded Shell . . . 578
17. Build Tool Plugins . . . 579
17.1. Spring Boot Maven Plugin . . . 579
17.2. Spring Boot Gradle Plugin . . . 579
17.3. Spring Boot AntLib Module . . . 579
17.3.1. Spring Boot Ant Tasks . . . 580
Using the “exejar” Task . . . 580
Examples . . . 580
17.3.2. Using the “findmainclass” Task . . . 581
Examples . . . 581
17.4. Supporting Other Build Systems . . . 581
17.4.1. Repackaging Archives . . . 582
17.4.2. Nested Libraries . . . 582
17.4.3. Finding a Main Class . . . 582

17.4.4. Example Repackage Implementation . . . 582
17.5. What to Read Next . . . 584
18. “How-to” Guides . . . 585
18.1. Spring Boot Application . . . 585
18.1.1. Create Your Own FailureAnalyzer . . . 585
18.1.2. Troubleshoot Auto-configuration . . . 585
18.1.3. Customize the Environment or ApplicationContext Before It Starts . . . 586
18.1.4. Build an ApplicationContext Hierarchy (Adding a Parent or Root Context) . . . 589
18.1.5. Create a Non-web Application . . . 589
18.2. Properties and Configuration . . . 589
18.2.1. Automatically Expand Properties at Build Time . . . 589
Automatic Property Expansion Using Maven . . . 589
Automatic Property Expansion Using Gradle . . . 590
18.2.2. Externalize the Configuration of SpringApplication . . . 591
18.2.3. Change the Location of External Properties of an Application . . . 594
18.2.4. Use ‘Short’ Command Line Arguments . . . 594
18.2.5. Use YAML for External Properties . . . 595
18.2.6. Set the Active Spring Profiles . . . 595
18.2.7. Set the Default Profile Name . . . 596
18.2.8. Change Configuration Depending on the Environment . . . 596
18.2.9. Discover Built-in Options for External Properties . . . 597
18.3. Embedded Web Servers . . . 598
18.3.1. Use Another Web Server . . . 598
18.3.2. Disabling the Web Server . . . 599
18.3.3. Change the HTTP Port . . . 600
18.3.4. Use a Random Unassigned HTTP Port . . . 600
18.3.5. Discover the HTTP Port at Runtime . . . 600
18.3.6. Enable HTTP Response Compression . . . 601
18.3.7. Configure SSL . . . 602
Using PEM-encoded files . . . 602
18.3.8. Configure HTTP/2 . . . 603
HTTP/2 With Tomcat . . . 603
HTTP/2 With Jetty . . . 604
HTTP/2 With Reactor Netty . . . 604
HTTP/2 With Undertow . . . 604
18.3.9. Configure the Web Server . . . 604
18.3.10. Add a Servlet, Filter, or Listener to an Application . . . 606
Add a Servlet, Filter, or Listener by Using a Spring Bean . . . 606
Add Servlets, Filters, and Listeners by Using Classpath Scanning . . . 607
18.3.11. Configure Access Logging . . . 607
18.3.12. Running Behind a Front-end Proxy Server . . . 609

Customize Tomcat's Proxy Configuration . . . 610
18.3.13. Enable Multiple Connectors with Tomcat . . . 610
18.3.14. Enable Tomcat's MBean Registry . . . 612
18.3.15. Enable Multiple Listeners with Undertow . . . 613
18.3.16. Create WebSocket Endpoints Using @ServerEndpoint . . . 614
18.4. Spring MVC . . . 615
18.4.1. Write a JSON REST Service . . . 615
18.4.2. Write an XML REST Service . . . 616
18.4.3. Customize the Jackson ObjectMapper . . . 617
18.4.4. Customize the @ResponseBody Rendering . . . 619
18.4.5. Handling Multipart File Uploads . . . 619
18.4.6. Switch Off the Spring MVC DispatcherServlet . . . 620
18.4.7. Switch off the Default MVC Configuration . . . 620
18.4.8. Customize ViewResolvers . . . 620
18.5. Jersey . . . 621
18.5.1. Secure Jersey endpoints with Spring Security . . . 622
18.5.2. Use Jersey Alongside Another Web Framework . . . 622
18.6. HTTP Clients . . . 623
18.6.1. Configure RestTemplate to Use a Proxy . . . 623
18.6.2. Configure the TcpClient used by a Reactor Netty-based WebClient . . . 623
18.7. Logging . . . 625
18.7.1. Configure Logback for Logging . . . 626
Configure Logback for File-only Output . . . 627
18.7.2. Configure Log4j for Logging . . . 628
Use YAML or JSON to Configure Log4j 2 . . . 629
Use Composite Configuration to Configure Log4j 2 . . . 630
18.8. Data Access . . . 630
18.8.1. Configure a Custom DataSource . . . 630
18.8.2. Configure Two DataSources . . . 637
18.8.3. Use Spring Data Repositories . . . 643
18.8.4. Separate @Entity Definitions from Spring Configuration . . . 643
18.8.5. Configure JPA Properties . . . 644
18.8.6. Configure Hibernate Naming Strategy . . . 645
18.8.7. Configure Hibernate Second-Level Caching . . . 647
18.8.8. Use Dependency Injection in Hibernate Components . . . 648
18.8.9. Use a Custom EntityManagerFactory . . . 649
18.8.10. Using Multiple EntityManagerFactories . . . 649
18.8.11. Use a Traditional persistence.xml File . . . 653
18.8.12. Use Spring Data JPA and Mongo Repositories . . . 653
18.8.13. Customize Spring Data's Web Support . . . 654
18.8.14. Expose Spring Data Repositories as REST Endpoint . . . 654

18.8.15. Configure a Component that is Used by JPA . . . 654
18.8.16. Configure jOOQ with Two DataSources . . . 655
18.9. Database Initialization . . . 655
18.9.1. Initialize a Database Using JPA . . . 656
18.9.2. Initialize a Database Using Hibernate . . . 656
18.9.3. Initialize a Database Using Basic SQL Scripts . . . 656
18.9.4. Initialize a Spring Batch Database . . . 657
18.9.5. Use a Higher-level Database Migration Tool . . . 657
Execute Flyway Database Migrations on Startup . . . 658
Execute Liquibase Database Migrations on Startup . . . 659
Use Flyway for test-only migrations . . . 660
Use Liquibase for test-only migrations . . . 660
18.9.6. Depend Upon an Initialized Database . . . 661
Detect a Database Initializer . . . 661
Detect a Bean That Depends On Database Initialization . . . 662
18.10. NoSQL . . . 662
18.10.1. Use Jedis Instead of Lettuce . . . 662
18.11. Messaging . . . 663
18.11.1. Disable Transacted JMS Session . . . 663
18.12. Batch Applications . . . 664
18.12.1. Specifying a Batch Data Source . . . 664
18.12.2. Running Spring Batch Jobs on Startup . . . 664
18.12.3. Running From the Command Line . . . 665
18.12.4. Restarting a stopped or failed Job . . . 665
18.12.5. Storing the Job Repository . . . 665
18.13. Actuator . . . 665
18.13.1. Change the HTTP Port or Address of the Actuator Endpoints . . . 665
18.13.2. Customize the ‘whitelabel’ Error Page . . . 666
18.13.3. Customizing Sanitization . . . 666
18.13.4. Map Health Indicators to Micrometer Metrics . . . 666
18.14. Security . . . 668
18.14.1. Switch off the Spring Boot Security Configuration . . . 668
18.14.2. Change the UserDetailsService and Add User Accounts . . . 668
18.14.3. Enable HTTPS When Running behind a Proxy Server . . . 669
18.15. Hot Swapping . . . 670
18.15.1. Reload Static Content . . . 670
18.15.2. Reload Templates without Restarting the Container . . . 671
Thymeleaf Templates . . . 671
FreeMarker Templates . . . 671
Groovy Templates . . . 671
18.15.3. Fast Application Restarts . . . 671

18.15.4. Reload Java Classes without Restarting the Container . . . 671
18.16. Testing . . . 672
18.16.1. Testing With Spring Security . . . 672
18.16.2. Structure `@Configuration` classes for inclusion in slice tests . . . 673
18.17. Build . . . 675
18.17.1. Generate Build Information . . . 676
18.17.2. Generate Git Information . . . 676
18.17.3. Customize Dependency Versions . . . 677
18.17.4. Create an Executable JAR with Maven . . . 677
18.17.5. Use a Spring Boot Application as a Dependency . . . 678
18.17.6. Extract Specific Libraries When an Executable Jar Runs . . . 679
18.17.7. Create a Non-executable JAR with Exclusions . . . 680
18.17.8. Remote Debug a Spring Boot Application Started with Maven . . . 680
18.17.9. Build an Executable Archive From Ant without Using spring-boot-antlib . . . 680
18.18. Ahead-of-time processing . . . 681
18.18.1. Conditions . . . 682
18.19. Traditional Deployment . . . 683
18.19.1. Create a Deployable War File . . . 683
18.19.2. Convert an Existing Application to Spring Boot . . . 685
18.19.3. Deploying a WAR to WebLogic . . . 690
18.20. Docker Compose . . . 691
18.20.1. Customizing the JDBC URL . . . 691
18.20.2. Sharing services between multiple applications . . . 691
Appendices . . . 693
Appendix A: Common Application Properties . . . 693
.A.1. Core Properties . . . 693
.A.2. Cache Properties . . . 701
.A.3. Mail Properties . . . 702
.A.4. JSON Properties . . . 703
.A.5. Data Properties . . . 705
.A.6. Transaction Properties . . . 729
.A.7. Data Migration Properties . . . 731
.A.8. Integration Properties . . . 738
.A.9. Web Properties . . . 761
.A.10. Templating Properties . . . 771
.A.11. Server Properties . . . 777
.A.12. Security Properties . . . 790
.A.13. RSocket Properties . . . 792
.A.14. Actuator Properties . . . 794
.A.15. Devtools Properties . . . 825
.A.16. Docker Compose Properties . . . 827

.A.17. Testing Properties . . . 828
Appendix B: Configuration Metadata . . . 828
.B.1. Metadata Format . . . 828
Group Attributes . . . 830
Property Attributes . . . 831
Hint Attributes . . . 833
Repeated Metadata Items . . . 834
.B.2. Providing Manual Hints . . . 835
Value Hint . . . 835
Value Providers . . . 836
.B.3. Generating Your Own Metadata by Using the Annotation Processor . . . 842
Configuring the Annotation Processor . . . 842
Automatic Metadata Generation . . . 843
Adding Additional Metadata . . . 849
Appendix C: Auto-configuration Classes . . . 849
.C.1. spring-boot-autoconfigure . . . 849
.C.2. spring-boot-actuator-autoconfigure . . . 854
Appendix D: Test Auto-configuration Annotations . . . 857
.D.1. Test Slices . . . 857
Appendix E: The Executable Jar Format . . . 868
.E.1. Nested JARs . . . 868
The Executable Jar File Structure . . . 868
The Executable War File Structure . . . 869
Index Files . . . 870
Classpath Index . . . 870
Layer Index . . . 870
.E.2. Spring Boot's "JarFile" Class . . . 871
Compatibility With the Standard Java "JarFile" . . . 871
.E.3. Launching Executable Jars . . . 871
Launcher Manifest . . . 872
.E.4. PropertiesLauncher Features . . . 872
.E.5. Executable Jar Restrictions . . . 874
.E.6. Alternative Single Jar Solutions . . . 874
Appendix F: Dependency Versions . . . 874
.F.1. Managed Dependency Coordinates . . . 874
.F.2. Version Properties . . . 912

This document is also available as multiple HTML pages and as a single HTML page.

Chapter 1. Legal

Copyright © 2012-2023

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Chapter 2. Getting Help

If you have trouble with Spring Boot, we would like to help.

- Try the How-to documents. They provide solutions to the most common questions.
- Learn the Spring basics. Spring Boot builds on many other Spring projects. Check the spring.io web-site for a wealth of reference documentation. If you are starting out with Spring, try one of the guides.
- Ask a question. We monitor stackoverflow.com for questions tagged with `spring-boot`.
- Report bugs with Spring Boot at github.com/spring-projects/spring-boot/issues.

NOTE	All of Spring Boot is open source, including the documentation. If you find problems with the docs or if you want to improve them, please get involved.

Chapter 3. Documentation Overview

This section provides a brief overview of Spring Boot reference documentation. It serves as a map for the rest of the document.

The latest copy of this document is available at docs.spring.io/spring-boot/docs/current/reference/.

3.1. First Steps

If you are getting started with Spring Boot or 'Spring' in general, start with the following topics:

- **From scratch:** Overview | Requirements | Installation
- **Tutorial:** Part 1 | Part 2
- **Running your example:** Part 1 | Part 2

3.2. Upgrading From an Earlier Version

You should always ensure that you are running a supported version of Spring Boot.

Depending on the version that you are upgrading to, you can find some additional tips here:

- **From 1.x:** Upgrading from 1.x
- **To a new feature release:** Upgrading to New Feature Release
- **Spring Boot CLI:** Upgrading the Spring Boot CLI

3.3. Developing With Spring Boot

Ready to actually start using Spring Boot? We have you covered:

- **Build systems:** Maven | Gradle | Ant | Starters
- **Best practices:** Code Structure | @Configuration | @EnableAutoConfiguration | Beans and Dependency Injection
- **Running your code:** IDE | Packaged | Maven | Gradle
- **Packaging your app:** Production jars
- **Spring Boot CLI:** Using the CLI

3.4. Learning About Spring Boot Features

Need more details about Spring Boot's core features? The following content is for you:

- **Spring Application:** SpringApplication
- **External Configuration:** External Configuration
- **Profiles:** Profiles

- **Logging:** Logging

3.5. Web

If you develop Spring Boot web applications, take a look at the following content:

- **Servlet Web Applications:** Spring MVC, Jersey, Embedded Servlet Containers
- **Reactive Web Applications:** Spring Webflux, Embedded Servlet Containers
- **Graceful Shutdown:** Graceful Shutdown
- **Spring Security:** Default Security Configuration, Auto-configuration for OAuth2, SAML
- **Spring Session:** Auto-configuration for Spring Session
- **Spring HATEOAS:** Auto-configuration for Spring HATEOAS

3.6. Data

If your application deals with a datastore, you can see how to configure that here:

- **SQL:** Configuring a SQL Datastore, Embedded Database support, Connection pools, and more.
- **NOSQL:** Auto-configuration for NOSQL stores such as Redis, MongoDB, Neo4j, and others.

3.7. Messaging

If your application uses any messaging protocol, see one or more of the following sections:

- **JMS:** Auto-configuration for ActiveMQ and Artemis, Sending and Receiving messages through JMS
- **AMQP:** Auto-configuration for RabbitMQ
- **Kafka:** Auto-configuration for Spring Kafka
- **RSocket:** Auto-configuration for Spring Framework's RSocket Support
- **Spring Integration:** Auto-configuration for Spring Integration

3.8. IO

If your application needs IO capabilities, see one or more of the following sections:

- **Caching:** Caching support with EhCache, Hazelcast, Infinispan, and more
- **Quartz:** Quartz Scheduling
- **Mail:** Sending Email
- **Validation:** JSR-303 Validation
- **REST Clients:** Calling REST Services with RestTemplate and WebClient
- **Webservices:** Auto-configuration for Spring Web Services

- **JTA:** Distributed Transactions with JTA

3.9. Container Images

Spring Boot provides first-class support for building efficient container images. You can read more about it here:

- **Efficient Container Images:** Tips to optimize container images such as Docker images
- **Dockerfiles:** Building container images using dockerfiles
- **Cloud Native Buildpacks:** Support for Cloud Native Buildpacks with Maven and Gradle

3.10. Moving to Production

When you are ready to push your Spring Boot application to production, we have some tricks that you might like:

- **Management endpoints:** Overview
- **Connection options:** HTTP | JMX
- **Monitoring:** Metrics | Auditing | HTTP Exchanges | Process

3.11. GraalVM Native Images

Spring Boot applications can be converted into native executables using GraalVM. You can read more about our native image support here:

- **GraalVM Native Images:** Introduction | Key Differences with the JVM | Ahead-of-Time Processing
- **Getting Started:** Buildpacks | Native Build Tools
- **Testing:** JVM | Native Build Tools
- **Advanced Topics:** Nested Configuration Properties | Converting JARs | Known Limitations

3.12. Advanced Topics

Finally, we have a few topics for more advanced users:

- **Spring Boot Applications Deployment:** Cloud Deployment | OS Service
- **Build tool plugins:** Maven | Gradle
- **Appendix:** Application Properties | Configuration Metadata | Auto-configuration Classes | Test Auto-configuration Annotations | Executable Jars | Dependency Versions

Chapter 4. Getting Started

If you are getting started with Spring Boot, or “Spring” in general, start by reading this section. It answers the basic “what?”, “how?” and “why?” questions. It includes an introduction to Spring Boot, along with installation instructions. We then walk you through building your first Spring Boot application, discussing some core principles as we go.

4.1. Introducing Spring Boot

Spring Boot helps you to create stand-alone, production-grade Spring-based applications that you can run. We take an opinionated view of the Spring platform and third-party libraries, so that you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

You can use Spring Boot to create Java applications that can be started by using `java -jar` or more traditional war deployments.

Our primary goals are:

- Provide a radically faster and widely accessible getting-started experience for all Spring development.
- Be opinionated out of the box but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (such as embedded servers, security, metrics, health checks, and externalized configuration).
- Absolutely no code generation (when not targeting native image) and no requirement for XML configuration.

4.2. System Requirements

Spring Boot 3.1.6 requires Java 17 and is compatible up to and including Java 21. Spring Framework 6.0.14 or above is also required.

Explicit build support is provided for the following build tools:

Build Tool	Version
Maven	3.6.3 or later
Gradle	7.x (7.5 or later) and 8.x

4.2.1. Servlet Containers

Spring Boot supports the following embedded servlet containers:

Name	Servlet Version
Tomcat 10.1	6.0

Name	Servlet Version
Jetty 11.0	5.0
Undertow 2.3	6.0

You can also deploy Spring Boot applications to any servlet 5.0+ compatible container.

4.2.2. GraalVM Native Images

Spring Boot applications can be converted into a Native Image using GraalVM 22.3 or above.

Images can be created using the native build tools Gradle/Maven plugins or `native-image` tool provided by GraalVM. You can also create native images using the native-image Paketo buildpack.

The following versions are supported:

Name	Version
GraalVM Community	22.3
Native Build Tools	0.9.28

4.3. Installing Spring Boot

Spring Boot can be used with "classic" Java development tools or installed as a command line tool. Either way, you need Java SDK v17 or higher. Before you begin, you should check your current Java installation by using the following command:

```
$ java -version
```

If you are new to Java development or if you want to experiment with Spring Boot, you might want to try the Spring Boot CLI (Command Line Interface) first. Otherwise, read on for "classic" installation instructions.

4.3.1. Installation Instructions for the Java Developer

You can use Spring Boot in the same way as any standard Java library. To do so, include the appropriate `spring-boot-*.jar` files on your classpath. Spring Boot does not require any special tools integration, so you can use any IDE or text editor. Also, there is nothing special about a Spring Boot application, so you can run and debug a Spring Boot application as you would any other Java program.

Although you *could* copy Spring Boot jars, we generally recommend that you use a build tool that supports dependency management (such as Maven or Gradle).

Maven Installation

Spring Boot is compatible with Apache Maven 3.6.3 or later. If you do not already have Maven installed, you can follow the instructions at maven.apache.org.

TIP	On many operating systems, Maven can be installed with a package manager. If you use OSX Homebrew, try `brew install maven`. Ubuntu users can run `sudo apt-get install maven`. Windows users with Chocolatey can run `choco install maven` from an elevated (administrator) prompt.

Spring Boot dependencies use the `org.springframework.boot` group id. Typically, your Maven POM file inherits from the `spring-boot-starter-parent` project and declares dependencies to one or more "Starters". Spring Boot also provides an optional Maven plugin to create executable jars.

More details on getting started with Spring Boot and Maven can be found in the Getting Started section of the Maven plugin's reference guide.

Gradle Installation

Spring Boot is compatible with Gradle 7.x (7.5 or later) and 8.x. If you do not already have Gradle installed, you can follow the instructions at gradle.org.

Spring Boot dependencies can be declared by using the `org.springframework.boot group`. Typically, your project declares dependencies to one or more "Starters". Spring Boot provides a useful Gradle plugin that can be used to simplify dependency declarations and to create executable jars.

Gradle Wrapper

The Gradle Wrapper provides a nice way of "obtaining" Gradle when you need to build a project. It is a small script and library that you commit alongside your code to bootstrap the build process. See docs.gradle.org/current/userguide/gradle_wrapper.html for details.

More details on getting started with Spring Boot and Gradle can be found in the Getting Started section of the Gradle plugin's reference guide.

4.3.2. Installing the Spring Boot CLI

The Spring Boot CLI (Command Line Interface) is a command line tool that you can use to quickly prototype with Spring.

You do not need to use the CLI to work with Spring Boot, but it is a quick way to get a Spring application off the ground without an IDE.

Manual Installation

You can download the Spring CLI distribution from one of the following locations:

- spring-boot-cli-3.1.6-bin.zip
- spring-boot-cli-3.1.6-bin.tar.gz

Once downloaded, follow the INSTALL.txt instructions from the unpacked archive. In summary, there is a `spring` script (`spring.bat` for Windows) in a `bin/` directory in the `.zip` file. Alternatively, you can use `java -jar` with the `.jar` file (the script helps you to be sure that the classpath is set

correctly).

Installation with SDKMAN!

SDKMAN! (The Software Development Kit Manager) can be used for managing multiple versions of various binary SDKs, including Groovy and the Spring Boot CLI. Get SDKMAN! from sdkman.io and install Spring Boot by using the following commands:

```
$ sdk install springboot
$ spring --version
Spring CLI v3.1.6
```

If you develop features for the CLI and want access to the version you built, use the following commands:

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-
cli-3.1.6-bin/spring-3.1.6/
$ sdk default springboot dev
$ spring --version
Spring CLI v3.1.6
```

The preceding instructions install a local instance of `spring` called the `dev` instance. It points at your target build location, so every time you rebuild Spring Boot, `spring` is up-to-date.

You can see it by running the following command:

```
$ sdk ls springboot

================================================================================
Available Springboot Versions
================================================================================
> + dev
* 3.1.6

================================================================================
+ - local version
* - installed
> - currently in use
================================================================================
```

OSX Homebrew Installation

If you are on a Mac and use Homebrew, you can install the Spring Boot CLI by using the following commands:

```
$ brew tap spring-io/tap
$ brew install spring-boot
```

Homebrew installs `spring` to `/usr/local/bin`.

NOTE	If you do not see the formula, your installation of brew might be out-of-date. In that case, run `brew update` and try again.

MacPorts Installation

If you are on a Mac and use MacPorts, you can install the Spring Boot CLI by using the following command:

```
$ sudo port install spring-boot-cli
```

Command-line Completion

The Spring Boot CLI includes scripts that provide command completion for the BASH and zsh shells. You can `source` the script (also named `spring`) in any shell or put it in your personal or system-wide bash completion initialization. On a Debian system, the system-wide scripts are in `<installation location>/shell-completion/bash` and all scripts in that directory are executed when a new shell starts. For example, to run the script manually if you have installed by using SDKMAN!, use the following commands:

```
$ . ~/.sdkman/candidates/springboot/current/shell-completion/bash/spring
$ spring <HIT TAB HERE>
  grab  help  jar  run  test  version
```

NOTE	If you install the Spring Boot CLI by using Homebrew or MacPorts, the command-line completion scripts are automatically registered with your shell.

Windows Scoop Installation

If you are on a Windows and use Scoop, you can install the Spring Boot CLI by using the following commands:

```
> scoop bucket add extras
> scoop install springboot
```

Scoop installs `spring` to `~/scoop/apps/springboot/current/bin`.

NOTE	If you do not see the app manifest, your installation of scoop might be out-of-date. In that case, run `scoop update` and try again.

4.4. Developing Your First Spring Boot Application

This section describes how to develop a small “Hello World!” web application that highlights some of Spring Boot’s key features. You can choose between Maven or Gradle as the build system.

> The spring.io website contains many “Getting Started” guides that use Spring Boot. If you need to solve a specific problem, check there first.
>
> **TIP** You can shortcut the steps below by going to start.spring.io and choosing the "Web" starter from the dependencies searcher. Doing so generates a new project structure so that you can start coding right away. Check the start.spring.io user guide for more details.

4.4.1. Prerequisites

Before we begin, open a terminal and run the following commands to ensure that you have a valid version of Java installed:

```
$ java -version
openjdk version "17.0.4.1" 2022-08-12 LTS
OpenJDK Runtime Environment (build 17.0.4.1+1-LTS)
OpenJDK 64-Bit Server VM (build 17.0.4.1+1-LTS, mixed mode, sharing)
```

> **NOTE** This sample needs to be created in its own directory. Subsequent instructions assume that you have created a suitable directory and that it is your current directory.

Maven

If you want to use Maven, ensure that you have Maven installed:

```
$ mvn -v
Apache Maven 3.8.5 (3599d3414f046de2324203b78ddcf9b5e4388aa0)
Maven home: usr/Users/developer/tools/maven/3.8.5
Java version: 17.0.4.1, vendor: BellSoft, runtime:
/Users/developer/sdkman/candidates/java/17.0.4.1-librca
```

Gradle

If you want to use Gradle, ensure that you have Gradle installed:

```
$ gradle --version

------------------------------------------------------------
Gradle 8.1.1
------------------------------------------------------------

Build time:   2023-04-21 12:31:26 UTC
Revision:     1cf537a851c635c364a4214885f8b9798051175b

Kotlin:       1.8.10
Groovy:       3.0.15
Ant:          Apache Ant(TM) version 1.10.11 compiled on July 10 2021
JVM:          17.0.7 (BellSoft 17.0.7+7-LTS)
OS:           Linux 6.2.12-200.fc37.aarch64 aarch64
```

4.4.2. Setting up the project with Maven

We need to start by creating a Maven `pom.xml` file. The `pom.xml` is the recipe that is used to build your project. Open your favorite text editor and add the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.6</version>
    </parent>

    <!-- Additional lines to be added here... -->

</project>
```

The preceding listing should give you a working build. You can test it by running `mvn package` (for now, you can ignore the “jar will be empty - no content was marked for inclusion!” warning).

NOTE	At this point, you could import the project into an IDE (most modern Java IDEs include built-in support for Maven). For simplicity, we continue to use a plain text editor for this example.

4.4.3. Setting up the project with Gradle

We need to start by creating a Gradle `build.gradle` file. The `build.gradle` is the build script that is used to build your project. Open your favorite text editor and add the following:

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.1.6'
}

apply plugin: 'io.spring.dependency-management'

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
}
```

The preceding listing should give you a working build. You can test it by running `gradle classes`.

NOTE	At this point, you could import the project into an IDE (most modern Java IDEs include built-in support for Gradle). For simplicity, we continue to use a plain text editor for this example.

4.4.4. Adding Classpath Dependencies

Spring Boot provides a number of "Starters" that let you add jars to your classpath. "Starters" provide dependencies that you are likely to need when developing a specific type of application.

Maven

Most Spring Boot applications use the `spring-boot-starter-parent` in the `parent` section of the POM. The `spring-boot-starter-parent` is a special starter that provides useful Maven defaults. It also provides a `dependency-management` section so that you can omit `version` tags for "blessed" dependencies.

Since we are developing a web application, we add a `spring-boot-starter-web` dependency. Before that, we can look at what we currently have by running the following command:

```
$ mvn dependency:tree

[INFO] com.example:myproject:jar:0.0.1-SNAPSHOT
```

The `mvn dependency:tree` command prints a tree representation of your project dependencies. You can see that `spring-boot-starter-parent` provides no dependencies by itself. To add the necessary dependencies, edit your `pom.xml` and add the `spring-boot-starter-web` dependency immediately below the `parent` section:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

If you run `mvn dependency:tree` again, you see that there are now a number of additional dependencies, including the Tomcat web server and Spring Boot itself.

Gradle

Most Spring Boot applications use the `org.springframework.boot` Gradle plugin. This plugin provides useful defaults and Gradle tasks. The `io.spring.dependency-management` Gradle plugin provides dependency management so that you can omit `version` tags for "blessed" dependencies.

Since we are developing a web application, we add a `spring-boot-starter-web` dependency. Before that, we can look at what we currently have by running the following command:

```
$ gradle dependencies

> Task :dependencies

------------------------------------------------------------
Root project 'myproject'
------------------------------------------------------------
```

The `gradle dependencies` command prints a tree representation of your project dependencies. Right now, the project has no dependencies. To add the necessary dependencies, edit your `build.gradle` and add the `spring-boot-starter-web` dependency in the `dependencies` section:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
}
```

If you run `gradle dependencies` again, you see that there are now a number of additional dependencies, including the Tomcat web server and Spring Boot itself.

4.4.5. Writing the Code

To finish our application, we need to create a single Java file. By default, Maven and Gradle compile sources from `src/main/java`, so you need to create that directory structure and then add a file

named `src/main/java/MyApplication.java` to contain the following code:

Java

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class MyApplication {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@RestController
@SpringBootApplication
class MyApplication {

    @RequestMapping("/")
    fun home() = "Hello World!"

}

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

Although there is not much code here, quite a lot is going on. We step through the important parts in the next few sections.

The @RestController and @RequestMapping Annotations

The first annotation on our `MyApplication` class is `@RestController`. This is known as a *stereotype* annotation. It provides hints for people reading the code and for Spring that the class plays a specific role. In this case, our class is a web `@Controller`, so Spring considers it when handling incoming web requests.

The `@RequestMapping` annotation provides "routing" information. It tells Spring that any HTTP request with the `/` path should be mapped to the `home` method. The `@RestController` annotation tells Spring to render the resulting string directly back to the caller.

TIP	The `@RestController` and `@RequestMapping` annotations are Spring MVC annotations (they are not specific to Spring Boot). See the MVC section in the Spring Reference Documentation for more details.

The @SpringBootApplication Annotation

The second class-level annotation is `@SpringBootApplication`. This annotation is known as a *meta-annotation*, it combines `@SpringBootConfiguration`, `@EnableAutoConfiguration` and `@ComponentScan`.

Of those, the annotation we're most interested in here is `@EnableAutoConfiguration`. `@EnableAutoConfiguration` tells Spring Boot to "guess" how you want to configure Spring, based on the jar dependencies that you have added. Since `spring-boot-starter-web` added Tomcat and Spring MVC, the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

Starters and Auto-configuration

Auto-configuration is designed to work well with "Starters", but the two concepts are not directly tied. You are free to pick and choose jar dependencies outside of the starters. Spring Boot still does its best to auto-configure your application.

The "main" Method

The final part of our application is the `main` method. This is a standard method that follows the Java convention for an application entry point. Our main method delegates to Spring Boot's `SpringApplication` class by calling `run`. `SpringApplication` bootstraps our application, starting Spring, which, in turn, starts the auto-configured Tomcat web server. We need to pass `MyApplication.class` as an argument to the `run` method to tell `SpringApplication` which is the primary Spring component. The `args` array is also passed through to expose any command-line arguments.

4.4.6. Running the Example

Maven

At this point, your application should work. Since you used the `spring-boot-starter-parent` POM, you have a useful `run` goal that you can use to start the application. Type `mvn spring-boot:run` from the root project directory to start the application. You should see output similar to the following:

```
$ mvn spring-boot:run


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v3.1.6)
....... . . .
....... . . . (log output here)
....... . . .
........ Started MyApplication in 0.906 seconds (process running for 6.514)
```

If you open a web browser to `localhost:8080`, you should see the following output:

```
Hello World!
```

To gracefully exit the application, press `ctrl-c`.

Gradle

At this point, your application should work. Since you used the `org.springframework.boot` Gradle plugin, you have a useful `bootRun` goal that you can use to start the application. Type `gradle bootRun` from the root project directory to start the application. You should see output similar to the following:

```
$ gradle bootRun


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v3.1.6)
....... . . .
....... . . . (log output here)
....... . . .
........ Started MyApplication in 0.906 seconds (process running for 6.514)
```

If you open a web browser to `localhost:8080`, you should see the following output:

```
Hello World!
```

To gracefully exit the application, press `ctrl-c`.

4.4.7. Creating an Executable Jar

We finish our example by creating a completely self-contained executable jar file that we could run in production. Executable jars (sometimes called “fat jars”) are archives containing your compiled classes along with all of the jar dependencies that your code needs to run.

Executable jars and Java

Java does not provide a standard way to load nested jar files (jar files that are themselves contained within a jar). This can be problematic if you are looking to distribute a self-contained application.

To solve this problem, many developers use “uber” jars. An uber jar packages all the classes from all the application’s dependencies into a single archive. The problem with this approach is that it becomes hard to see which libraries are in your application. It can also be problematic if the same filename is used (but with different content) in multiple jars.

Spring Boot takes a different approach and lets you actually nest jars directly.

Maven

To create an executable jar, we need to add the `spring-boot-maven-plugin` to our `pom.xml`. To do so, insert the following lines just below the `dependencies` section:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

NOTE The `spring-boot-starter-parent` POM includes `<executions>` configuration to bind the `repackage` goal. If you do not use the parent POM, you need to declare this configuration yourself. See the plugin documentation for details.

Save your `pom.xml` and run `mvn package` from the command line, as follows:

```
$ mvn package

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building myproject 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] .... ..
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ myproject ---
[INFO] Building jar: /Users/developer/example/spring-boot-example/target/myproject-
0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:3.1.6:repackage (default) @ myproject ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

If you look in the `target` directory, you should see `myproject-0.0.1-SNAPSHOT.jar`. The file should be around 18 MB in size. If you want to peek inside, you can use `jar tvf`, as follows:

```
$ jar tvf target/myproject-0.0.1-SNAPSHOT.jar
```

You should also see a much smaller file named `myproject-0.0.1-SNAPSHOT.jar.original` in the `target` directory. This is the original jar file that Maven created before it was repackaged by Spring Boot.

To run that application, use the `java -jar` command, as follows:

```
$ java -jar target/myproject-0.0.1-SNAPSHOT.jar


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v3.1.6)
....... . . .
....... . . . (log output here)
....... . . .
........ Started MyApplication in 0.999 seconds (process running for 1.253)
```

As before, to exit the application, press `ctrl-c`.

Gradle

To create an executable jar, we need to run `gradle bootJar` from the command line, as follows:

```
$ gradle bootJar

BUILD SUCCESSFUL in 639ms
3 actionable tasks: 3 executed
```

If you look in the `build/libs` directory, you should see `myproject-0.0.1-SNAPSHOT.jar`. The file should be around 18 MB in size. If you want to peek inside, you can use `jar tvf`, as follows:

```
$ jar tvf build/libs/myproject-0.0.1-SNAPSHOT.jar
```

To run that application, use the `java -jar` command, as follows:

```
$ java -jar build/libs/myproject-0.0.1-SNAPSHOT.jar


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v3.1.6)
....... . . .
....... . . . (log output here)
....... . . .
........ Started MyApplication in 0.999 seconds (process running for 1.253)
```

As before, to exit the application, press `ctrl-c`.

4.5. What to Read Next

Hopefully, this section provided some of the Spring Boot basics and got you on your way to writing your own applications. If you are a task-oriented type of developer, you might want to jump over to spring.io and follow some of the getting started guides that solve specific "How do I do that with Spring?" problems. We also have Spring Boot-specific "How-to" reference documentation.

Otherwise, the next logical step is to read *Developing with Spring Boot*. If you are really impatient, you could also jump ahead and read about *Spring Boot features*.

Chapter 5. Upgrading Spring Boot

Instructions for how to upgrade from earlier versions of Spring Boot are provided on the project wiki. Follow the links in the release notes section to find the version that you want to upgrade to.

Upgrading instructions are always the first item in the release notes. If you are more than one release behind, please make sure that you also review the release notes of the versions that you jumped.

5.1. Upgrading From 1.x

If you are upgrading from the `1.x` release of Spring Boot, check the "migration guide" on the project wiki that provides detailed upgrade instructions. Check also the "release notes" for a list of "new and noteworthy" features for each release.

5.2. Upgrading to a New Feature Release

When upgrading to a new feature release, some properties may have been renamed or removed. Spring Boot provides a way to analyze your application's environment and print diagnostics at startup, but also temporarily migrate properties at runtime for you. To enable that feature, add the following dependency to your project:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-properties-migrator</artifactId>
    <scope>runtime</scope>
</dependency>
```

WARNING Properties that are added late to the environment, such as when using `@PropertySource`, will not be taken into account.

NOTE Once you finish the migration, please make sure to remove this module from your project's dependencies.

5.3. Upgrading the Spring Boot CLI

To upgrade an existing CLI installation, use the appropriate package manager command (for example, `brew upgrade`). If you manually installed the CLI, follow the standard instructions, remembering to update your `PATH` environment variable to remove any older references.

5.4. What to Read Next

Once you've decided to upgrade your application, you can find detailed information regarding specific features in the rest of the document.

Spring Boot's documentation is specific to that version, so any information that you find in here will contain the most up-to-date changes that are in that version.

Chapter 6. Developing with Spring Boot

This section goes into more detail about how you should use Spring Boot. It covers topics such as build systems, auto-configuration, and how to run your applications. We also cover some Spring Boot best practices. Although there is nothing particularly special about Spring Boot (it is just another library that you can consume), there are a few recommendations that, when followed, make your development process a little easier.

If you are starting out with Spring Boot, you should probably read the *Getting Started* guide before diving into this section.

6.1. Build Systems

It is strongly recommended that you choose a build system that supports *dependency management* and that can consume artifacts published to the "Maven Central" repository. We would recommend that you choose Maven or Gradle. It is possible to get Spring Boot to work with other build systems (Ant, for example), but they are not particularly well supported.

6.1.1. Dependency Management

Each release of Spring Boot provides a curated list of dependencies that it supports. In practice, you do not need to provide a version for any of these dependencies in your build configuration, as Spring Boot manages that for you. When you upgrade Spring Boot itself, these dependencies are upgraded as well in a consistent way.

NOTE	You can still specify a version and override Spring Boot's recommendations if you need to do so.

The curated list contains all the Spring modules that you can use with Spring Boot as well as a refined list of third party libraries. The list is available as a standard Bills of Materials (`spring-boot-dependencies`) that can be used with both Maven and Gradle.

WARNING	Each release of Spring Boot is associated with a base version of the Spring Framework. We **highly** recommend that you do not specify its version.

6.1.2. Maven

To learn about using Spring Boot with Maven, see the documentation for Spring Boot's Maven plugin:

- Reference (HTML and PDF)
- API

6.1.3. Gradle

To learn about using Spring Boot with Gradle, see the documentation for Spring Boot's Gradle plugin:

- Reference (HTML and PDF)
- API

6.1.4. Ant

It is possible to build a Spring Boot project using Apache Ant+Ivy. The `spring-boot-antlib` "AntLib" module is also available to help Ant create executable jars.

To declare dependencies, a typical `ivy.xml` file looks something like the following example:

```
<ivy-module version="2.0">
    <info organisation="org.springframework.boot" module="spring-boot-sample-ant" />
    <configurations>
        <conf name="compile" description="everything needed to compile this module" />
        <conf name="runtime" extends="compile" description="everything needed to run
this module" />
    </configurations>
    <dependencies>
        <dependency org="org.springframework.boot" name="spring-boot-starter"
            rev="${spring-boot.version}" conf="compile" />
    </dependencies>
</ivy-module>
```

A typical `build.xml` looks like the following example:

```
<project
    xmlns:ivy="antlib:org.apache.ivy.ant"
    xmlns:spring-boot="antlib:org.springframework.boot.ant"
    name="myapp" default="build">

    <property name="spring-boot.version" value="3.1.6" />

    <target name="resolve" description="--> retrieve dependencies with ivy">
        <ivy:retrieve pattern="lib/[conf]/[artifact]-[type]-[revision].[ext]" />
    </target>

    <target name="classpaths" depends="resolve">
        <path id="compile.classpath">
            <fileset dir="lib/compile" includes="*.jar" />
        </path>
    </target>

    <target name="init" depends="classpaths">
        <mkdir dir="build/classes" />
    </target>

    <target name="compile" depends="init" description="compile">
        <javac srcdir="src/main/java" destdir="build/classes"
classpathref="compile.classpath" />
    </target>

    <target name="build" depends="compile">
        <spring-boot:exejar destfile="build/myapp.jar" classes="build/classes">
            <spring-boot:lib>
                <fileset dir="lib/runtime" />
            </spring-boot:lib>
        </spring-boot:exejar>
    </target>
</project>
```

TIP If you do not want to use the `spring-boot-antlib` module, see the *Build an Executable Archive From Ant without Using spring-boot-antlib* "How-to" .

6.1.5. Starters

Starters are a set of convenient dependency descriptors that you can include in your application. You get a one-stop shop for all the Spring and related technologies that you need without having to hunt through sample code and copy-paste loads of dependency descriptors. For example, if you want to get started using Spring and JPA for database access, include the `spring-boot-starter-data-jpa` dependency in your project.

The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.

What is in a name

All **official** starters follow a similar naming pattern; `spring-boot-starter-*`, where `*` is a particular type of application. This naming structure is intended to help when you need to find a starter. The Maven integration in many IDEs lets you search dependencies by name. For example, with the appropriate Eclipse or Spring Tools plugin installed, you can press `ctrl-space` in the POM editor and type "spring-boot-starter" for a complete list.

As explained in the "Creating Your Own Starter" section, third party starters should not start with `spring-boot`, as it is reserved for official Spring Boot artifacts. Rather, a third-party starter typically starts with the name of the project. For example, a third-party starter project called `thirdpartyproject` would typically be named `thirdpartyproject-spring-boot-starter`.

The following application starters are provided by Spring Boot under the `org.springframework.boot` group:

Table 1. Spring Boot application starters

Name	Description
`spring-boot-starter`	Core starter, including auto-configuration support, logging and YAML
`spring-boot-starter-activemq`	Starter for JMS messaging using Apache ActiveMQ
`spring-boot-starter-amqp`	Starter for using Spring AMQP and Rabbit MQ
`spring-boot-starter-aop`	Starter for aspect-oriented programming with Spring AOP and AspectJ
`spring-boot-starter-artemis`	Starter for JMS messaging using Apache Artemis
`spring-boot-starter-batch`	Starter for using Spring Batch
`spring-boot-starter-cache`	Starter for using Spring Framework's caching support
`spring-boot-starter-data-cassandra`	Starter for using Cassandra distributed database and Spring Data Cassandra
`spring-boot-starter-data-cassandra-reactive`	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive
`spring-boot-starter-data-couchbase`	Starter for using Couchbase document-oriented database and Spring Data Couchbase
`spring-boot-starter-data-couchbase-reactive`	Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive
`spring-boot-starter-data-elasticsearch`	Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch
`spring-boot-starter-data-jdbc`	Starter for using Spring Data JDBC
`spring-boot-starter-data-jpa`	Starter for using Spring Data JPA with Hibernate

Name	Description
`spring-boot-starter-data-ldap`	Starter for using Spring Data LDAP
`spring-boot-starter-data-mongodb`	Starter for using MongoDB document-oriented database and Spring Data MongoDB
`spring-boot-starter-data-mongodb-reactive`	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive
`spring-boot-starter-data-neo4j`	Starter for using Neo4j graph database and Spring Data Neo4j
`spring-boot-starter-data-r2dbc`	Starter for using Spring Data R2DBC
`spring-boot-starter-data-redis`	Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client
`spring-boot-starter-data-redis-reactive`	Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client
`spring-boot-starter-data-rest`	Starter for exposing Spring Data repositories over REST using Spring Data REST and Spring MVC
`spring-boot-starter-freemarker`	Starter for building MVC web applications using FreeMarker views
`spring-boot-starter-graphql`	Starter for building GraphQL applications with Spring GraphQL
`spring-boot-starter-groovy-templates`	Starter for building MVC web applications using Groovy Templates views
`spring-boot-starter-hateoas`	Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS
`spring-boot-starter-integration`	Starter for using Spring Integration
`spring-boot-starter-jdbc`	Starter for using JDBC with the HikariCP connection pool
`spring-boot-starter-jersey`	Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to `spring-boot-starter-web`
`spring-boot-starter-jooq`	Starter for using jOOQ to access SQL databases with JDBC. An alternative to `spring-boot-starter-data-jpa` or `spring-boot-starter-jdbc`
`spring-boot-starter-json`	Starter for reading and writing json
`spring-boot-starter-mail`	Starter for using Java Mail and Spring Framework's email sending support
`spring-boot-starter-mustache`	Starter for building web applications using Mustache views

Name	Description
spring-boot-starter-oauth2-authorization-server	Starter for using Spring Authorization Server features
spring-boot-starter-oauth2-client	Starter for using Spring Security's OAuth2/OpenID Connect client features
spring-boot-starter-oauth2-resource-server	Starter for using Spring Security's OAuth2 resource server features
spring-boot-starter-quartz	Starter for using the Quartz scheduler
spring-boot-starter-rsocket	Starter for building RSocket clients and servers
spring-boot-starter-security	Starter for using Spring Security
spring-boot-starter-test	Starter for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest and Mockito
spring-boot-starter-thymeleaf	Starter for building MVC web applications using Thymeleaf views
spring-boot-starter-validation	Starter for using Java Bean Validation with Hibernate Validator
spring-boot-starter-web	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
spring-boot-starter-web-services	Starter for using Spring Web Services
spring-boot-starter-webflux	Starter for building WebFlux applications using Spring Framework's Reactive Web support
spring-boot-starter-websocket	Starter for building WebSocket applications using Spring Framework's MVC WebSocket support

In addition to the application starters, the following starters can be used to add *production ready* features:

Table 2. Spring Boot production starters

Name	Description
spring-boot-starter-actuator	Starter for using Spring Boot's Actuator which provides production ready features to help you monitor and manage your application

Finally, Spring Boot also includes the following starters that can be used if you want to exclude or swap specific technical facets:

Table 3. Spring Boot technical starters

Name	Description
`spring-boot-starter-jetty`	Starter for using Jetty as the embedded servlet container. An alternative to `spring-boot-starter-tomcat`
`spring-boot-starter-log4j2`	Starter for using Log4j2 for logging. An alternative to `spring-boot-starter-logging`
`spring-boot-starter-logging`	Starter for logging using Logback. Default logging starter
`spring-boot-starter-reactor-netty`	Starter for using Reactor Netty as the embedded reactive HTTP server.
`spring-boot-starter-tomcat`	Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by `spring-boot-starter-web`
`spring-boot-starter-undertow`	Starter for using Undertow as the embedded servlet container. An alternative to `spring-boot-starter-tomcat`

To learn how to swap technical facets, please see the how-to documentation for swapping web server and logging system.

TIP For a list of additional community contributed starters, see the README file in the `spring-boot-starters` module on GitHub.

6.2. Structuring Your Code

Spring Boot does not require any specific code layout to work. However, there are some best practices that help.

6.2.1. Using the "default" Package

When a class does not include a `package` declaration, it is considered to be in the "default package". The use of the "default package" is generally discouraged and should be avoided. It can cause particular problems for Spring Boot applications that use the `@ComponentScan`, `@ConfigurationPropertiesScan`, `@EntityScan`, or `@SpringBootApplication` annotations, since every class from every jar is read.

TIP We recommend that you follow Java's recommended package naming conventions and use a reversed domain name (for example, `com.example.project`).

6.2.2. Locating the Main Application Class

We generally recommend that you locate your main application class in a root package above other classes. The `@SpringBootApplication` annotation is often placed on your main class, and it implicitly defines a base "search package" for certain items. For example, if you are writing a JPA application, the package of the `@SpringBootApplication` annotated class is used to search for `@Entity` items. Using

a root package also allows component scan to apply only on your project.

TIP	If you do not want to use `@SpringBootApplication`, the `@EnableAutoConfiguration` and `@ComponentScan` annotations that it imports defines that behavior so you can also use those instead.

The following listing shows a typical layout:

```
com
 +- example
     +- myapplication
         +- MyApplication.java
         |
         +- customer
         |   +- Customer.java
         |   +- CustomerController.java
         |   +- CustomerService.java
         |   +- CustomerRepository.java
         |
         +- order
             +- Order.java
             +- OrderController.java
             +- OrderService.java
             +- OrderRepository.java
```

The `MyApplication.java` file would declare the `main` method, along with the basic `@SpringBootApplication`, as follows:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

6.3. Configuration Classes

Spring Boot favors Java-based configuration. Although it is possible to use `SpringApplication` with XML sources, we generally recommend that your primary source be a single `@Configuration` class. Usually the class that defines the `main` method is a good candidate as the primary `@Configuration`.

TIP Many Spring configuration examples have been published on the Internet that use XML configuration. If possible, always try to use the equivalent Java-based configuration. Searching for `Enable*` annotations can be a good starting point.

6.3.1. Importing Additional Configuration Classes

You need not put all your `@Configuration` into a single class. The `@Import` annotation can be used to import additional configuration classes. Alternatively, you can use `@ComponentScan` to automatically pick up all Spring components, including `@Configuration` classes.

6.3.2. Importing XML Configuration

If you absolutely must use XML based configuration, we recommend that you still start with a `@Configuration` class. You can then use an `@ImportResource` annotation to load XML configuration files.

6.4. Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, if `HSQLDB` is on your classpath, and you have not manually configured any database connection beans, then Spring Boot auto-configures an in-memory database.

You need to opt-in to auto-configuration by adding the `@EnableAutoConfiguration` or `@SpringBootApplication` annotations to one of your `@Configuration` classes.

TIP You should only ever add one `@SpringBootApplication` or `@EnableAutoConfiguration` annotation. We generally recommend that you add one or the other to your primary `@Configuration` class only.

6.4.1. Gradually Replacing Auto-configuration

Auto-configuration is non-invasive. At any point, you can start to define your own configuration to replace specific parts of the auto-configuration. For example, if you add your own `DataSource` bean, the default embedded database support backs away.

If you need to find out what auto-configuration is currently being applied, and why, start your application with the `--debug` switch. Doing so enables debug logs for a selection of core loggers and logs a conditions report to the console.

6.4.2. Disabling Specific Auto-configuration Classes

If you find that specific auto-configuration classes that you do not want are being applied, you can use the exclude attribute of `@SpringBootApplication` to disable them, as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;

@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
public class MyApplication {

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

@SpringBootApplication(exclude = [DataSourceAutoConfiguration::class])
class MyApplication
```

If the class is not on the classpath, you can use the `excludeName` attribute of the annotation and specify the fully qualified name instead. If you prefer to use `@EnableAutoConfiguration` rather than `@SpringBootApplication`, `exclude` and `excludeName` are also available. Finally, you can also control the list of auto-configuration classes to exclude by using the `spring.autoconfigure.exclude` property.

TIP	You can define exclusions both at the annotation level and by using the property.
NOTE	Even though auto-configuration classes are `public`, the only aspect of the class that is considered public API is the name of the class which can be used for disabling the auto-configuration. The actual contents of those classes, such as nested configuration classes or bean methods are for internal use only and we do not recommend using those directly.

6.4.3. Auto-configuration Packages

Auto-configuration packages are the packages that various auto-configured features look in by default when scanning for things such as entities and Spring Data repositories. The `@EnableAutoConfiguration` annotation (either directly or through its presence on `@SpringBootApplication`) determines the default auto-configuration package. Additional packages can be configured using the `@AutoConfigurationPackage` annotation.

6.5. Spring Beans and Dependency Injection

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. We generally recommend using constructor injection to wire up dependencies and `@ComponentScan` to find beans.

If you structure your code as suggested above (locating your application class in a top package), you can add `@ComponentScan` without any arguments or use the `@SpringBootApplication` annotation which implicitly includes it. All of your application components (`@Component`, `@Service`, `@Repository`, `@Controller`, and others) are automatically registered as Spring Beans.

The following example shows a `@Service` Bean that uses constructor injection to obtain a required `RiskAssessor` bean:

Java

```
import org.springframework.stereotype.Service;

@Service
public class MyAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    public MyAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
    }

    // ...

}
```

Kotlin

```
import org.springframework.stereotype.Service

@Service
class MyAccountService(private val riskAssessor: RiskAssessor) : AccountService
```

If a bean has more than one constructor, you will need to mark the one you want Spring to use with `@Autowired`:

Java

```
import java.io.PrintStream;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class MyAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    private final PrintStream out;

    @Autowired
    public MyAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
        this.out = System.out;
    }

    public MyAccountService(RiskAssessor riskAssessor, PrintStream out) {
        this.riskAssessor = riskAssessor;
        this.out = out;
    }

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service
import java.io.PrintStream

@Service
class MyAccountService : AccountService {

    private val riskAssessor: RiskAssessor

    private val out: PrintStream

    @Autowired
    constructor(riskAssessor: RiskAssessor) {
        this.riskAssessor = riskAssessor
        out = System.out
    }

    constructor(riskAssessor: RiskAssessor, out: PrintStream) {
        this.riskAssessor = riskAssessor
        this.out = out
    }

    // ...

}
```

TIP Notice how using constructor injection lets the `riskAssessor` field be marked as `final`, indicating that it cannot be subsequently changed.

6.6. Using the @SpringBootApplication Annotation

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single `@SpringBootApplication` annotation can be used to enable those three features, that is:

- `@EnableAutoConfiguration`: enable Spring Boot's auto-configuration mechanism
- `@ComponentScan`: enable `@Component` scan on the package where the application is located (see the best practices)
- `@SpringBootConfiguration`: enable registration of extra beans in the context or the import of additional configuration classes. An alternative to Spring's standard `@Configuration` that aids configuration detection in your integration tests.

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

// Same as @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan
@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

// same as @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan
@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

NOTE `@SpringBootApplication` also provides aliases to customize the attributes of `@EnableAutoConfiguration` and `@ComponentScan`.

None of these features are mandatory and you may choose to replace this single annotation by any of the features that it enables. For instance, you may not want to use component scan or configuration properties scan in your application:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.SpringBootConfiguration;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.Import;

@SpringBootConfiguration(proxyBeanMethods = false)
@EnableAutoConfiguration
@Import({ SomeConfiguration.class, AnotherConfiguration.class })
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

NOTE

Kotlin

```
import org.springframework.boot.SpringBootConfiguration
import org.springframework.boot.autoconfigure.EnableAutoConfiguration
import
org.springframework.boot.docs.using.structuringyourcode.locatingthemainc
lass.MyApplication
import org.springframework.boot.runApplication
import org.springframework.context.annotation.Import

@SpringBootConfiguration(proxyBeanMethods = false)
@EnableAutoConfiguration
@Import(SomeConfiguration::class, AnotherConfiguration::class)
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

In this example, `MyApplication` is just like any other Spring Boot application except that `@Component`-annotated classes and `@ConfigurationProperties`-annotated classes are not detected automatically and the user-defined beans are imported explicitly (see `@Import`).

6.7. Running Your Application

One of the biggest advantages of packaging your application as a jar and using an embedded HTTP server is that you can run your application as you would any other. The sample applies to debugging Spring Boot applications. You do not need any special IDE plugins or extensions.

NOTE	This section only covers jar-based packaging. If you choose to package your application as a war file, see your server and IDE documentation.

6.7.1. Running From an IDE

You can run a Spring Boot application from your IDE as a Java application. However, you first need to import your project. Import steps vary depending on your IDE and build system. Most IDEs can import Maven projects directly. For example, Eclipse users can select `Import… → Existing Maven Projects` from the `File` menu.

If you cannot directly import your project into your IDE, you may be able to generate IDE metadata by using a build plugin. Maven includes plugins for Eclipse and IDEA. Gradle offers plugins for various IDEs.

TIP	If you accidentally run a web application twice, you see a “Port already in use” error. Spring Tools users can use the `Relaunch` button rather than the `Run` button to ensure that any existing instance is closed.

6.7.2. Running as a Packaged Application

If you use the Spring Boot Maven or Gradle plugins to create an executable jar, you can run your application using `java -jar`, as shown in the following example:

```
$ java -jar target/myapplication-0.0.1-SNAPSHOT.jar
```

It is also possible to run a packaged application with remote debugging support enabled. Doing so lets you attach a debugger to your packaged application, as shown in the following example:

```
$ java -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000,suspend=n \
       -jar target/myapplication-0.0.1-SNAPSHOT.jar
```

6.7.3. Using the Maven Plugin

The Spring Boot Maven plugin includes a `run` goal that can be used to quickly compile and run your application. Applications run in an exploded form, as they do in your IDE. The following example shows a typical Maven command to run a Spring Boot application:

```
$ mvn spring-boot:run
```

You might also want to use the `MAVEN_OPTS` operating system environment variable, as shown in the following example:

```
$ export MAVEN_OPTS=-Xmx1024m
```

6.7.4. Using the Gradle Plugin

The Spring Boot Gradle plugin also includes a `bootRun` task that can be used to run your application in an exploded form. The `bootRun` task is added whenever you apply the `org.springframework.boot` and `java` plugins and is shown in the following example:

```
$ gradle bootRun
```

You might also want to use the `JAVA_OPTS` operating system environment variable, as shown in the following example:

```
$ export JAVA_OPTS=-Xmx1024m
```

6.7.5. Hot Swapping

Since Spring Boot applications are plain Java applications, JVM hot-swapping should work out of the box. JVM hot swapping is somewhat limited with the bytecode that it can replace. For a more complete solution, JRebel can be used.

The `spring-boot-devtools` module also includes support for quick application restarts. See the Hot swapping "How-to" for details.

6.8. Developer Tools

Spring Boot includes an additional set of tools that can make the application development experience a little more pleasant. The `spring-boot-devtools` module can be included in any project to provide additional development-time features. To include devtools support, add the module dependency to your build, as shown in the following listings for Maven and Gradle:

Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

Gradle

```
dependencies {
    developmentOnly("org.springframework.boot:spring-boot-devtools")
}
```

CAUTION	Devtools might cause classloading issues, in particular in multi-module projects. Diagnosing Classloading Issues explains how to diagnose and solve them.
NOTE	Developer tools are automatically disabled when running a fully packaged application. If your application is launched from `java -jar` or if it is started from a special classloader, then it is considered a "production application". You can control this behavior by using the `spring.devtools.restart.enabled` system property. To enable devtools, irrespective of the classloader used to launch your application, set the `-Dspring.devtools.restart.enabled=true` system property. This must not be done in a production environment where running devtools is a security risk. To disable devtools, exclude the dependency or set the `-Dspring.devtools.restart.enabled=false` system property.
TIP	Flagging the dependency as optional in Maven or using the `developmentOnly` configuration in Gradle (as shown above) prevents devtools from being transitively applied to other modules that use your project.
TIP	Repackaged archives do not contain devtools by default. If you want to use a certain remote devtools feature, you need to include it. When using the Maven plugin, set the `excludeDevtools` property to `false`. When using the Gradle plugin, configure the task's classpath to include the `developmentOnly` configuration.

6.8.1. Diagnosing Classloading Issues

As described in the Restart vs Reload section, restart functionality is implemented by using two classloaders. For most applications, this approach works well. However, it can sometimes cause classloading issues, in particular in multi-module projects.

To diagnose whether the classloading issues are indeed caused by devtools and its two classloaders, try disabling restart. If this solves your problems, customize the restart classloader to include your entire project.

6.8.2. Property Defaults

Several of the libraries supported by Spring Boot use caches to improve performance. For example, template engines cache compiled templates to avoid repeatedly parsing template files. Also, Spring MVC can add HTTP caching headers to responses when serving static resources.

While caching is very beneficial in production, it can be counter-productive during development, preventing you from seeing the changes you just made in your application. For this reason, spring-boot-devtools disables the caching options by default.

Cache options are usually configured by settings in your `application.properties` file. For example, Thymeleaf offers the `spring.thymeleaf.cache` property. Rather than needing to set these properties manually, the `spring-boot-devtools` module automatically applies sensible development-time configuration.

The following table lists all the properties that are applied:

Name	Default Value
`server.error.include-binding-errors`	`always`
`server.error.include-message`	`always`
`server.error.include-stacktrace`	`always`
`server.servlet.jsp.init-parameters.development`	`true`
`server.servlet.session.persistent`	`true`
`spring.docker.compose.readiness.wait`	`only-if-started`
`spring.freemarker.cache`	`false`
`spring.graphql.graphiql.enabled`	`true`
`spring.groovy.template.cache`	`false`
`spring.h2.console.enabled`	`true`
`spring.mustache.servlet.cache`	`false`
`spring.mvc.log-resolved-exception`	`true`
`spring.reactor.netty.shutdown-quiet-period`	`0s`
`spring.template.provider.cache`	`false`
`spring.thymeleaf.cache`	`false`
`spring.web.resources.cache.period`	`0`
`spring.web.resources.chain.cache`	`false`

NOTE If you do not want property defaults to be applied you can set `spring.devtools.add-properties` to `false` in your `application.properties`.

Because you need more information about web requests while developing Spring MVC and Spring WebFlux applications, developer tools suggests you to enable `DEBUG` logging for the `web` logging group. This will give you information about the incoming request, which handler is processing it, the response outcome, and other details. If you wish to log all request details (including potentially sensitive information), you can turn on the `spring.mvc.log-request-details` or `spring.codec.log-request-details` configuration properties.

6.8.3. Automatic Restart

Applications that use `spring-boot-devtools` automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE, as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a directory is monitored for changes. Note that certain resources, such as static assets and view templates, do not need to restart the application.

Triggering a restart

As DevTools monitors classpath resources, the only way to trigger a restart is to update the classpath. Whether you're using an IDE or one of the build plugins, the modified files have to be recompiled to trigger a restart. The way in which you cause the classpath to be updated depends on the tool that you are using:

- In Eclipse, saving a modified file causes the classpath to be updated and triggers a restart.
- In IntelliJ IDEA, building the project (`Build +↑+ Build Project`) has the same effect.
- If using a build plugin, running `mvn compile` for Maven or `gradle build` for Gradle will trigger a restart.

NOTE If you are restarting with Maven or Gradle using the build plugin you must leave the `forking` set to `enabled`. If you disable forking, the isolated application classloader used by devtools will not be created and restarts will not operate properly.

TIP Automatic restart works very well when used with LiveReload. See the LiveReload section for details. If you use JRebel, automatic restarts are disabled in favor of dynamic class reloading. Other devtools features (such as LiveReload and property overrides) can still be used.

NOTE DevTools relies on the application context's shutdown hook to close it during a restart. It does not work correctly if you have disabled the shutdown hook (`SpringApplication.setRegisterShutdownHook(false)`).

NOTE DevTools needs to customize the `ResourceLoader` used by the `ApplicationContext`. If your application provides one already, it is going to be wrapped. Direct override of the `getResource` method on the `ApplicationContext` is not supported.

CAUTION Automatic restart is not supported when using AspectJ weaving.

Restart vs Reload

The restart technology provided by Spring Boot works by using two classloaders. Classes that do not change (for example, those from third-party jars) are loaded into a *base* classloader. Classes that you are actively developing are loaded into a *restart* classloader. When the application is restarted, the *restart* classloader is thrown away and a new one is created. This approach means that application restarts are typically much faster than "cold starts", since the *base* classloader is already available and populated.

If you find that restarts are not quick enough for your applications or you encounter classloading issues, you could consider reloading technologies such as JRebel from ZeroTurnaround. These work by rewriting classes as they are loaded to make them more amenable to reloading.

Logging Changes in Condition Evaluation

By default, each time your application restarts, a report showing the condition evaluation delta is logged. The report shows the changes to your application's auto-configuration as you make changes such as adding or removing beans and setting configuration properties.

To disable the logging of the report, set the following property:

Properties

```
spring.devtools.restart.log-condition-evaluation-delta=false
```

Yaml

```
spring:
  devtools:
    restart:
      log-condition-evaluation-delta: false
```

Excluding Resources

Certain resources do not necessarily need to trigger a restart when they are changed. For example, Thymeleaf templates can be edited in-place. By default, changing resources in `/META-INF/maven`, `/META-INF/resources`, `/resources`, `/static`, `/public`, or `/templates` does not trigger a restart but does trigger a live reload. If you want to customize these exclusions, you can use the `spring.devtools.restart.exclude` property. For example, to exclude only `/static` and `/public` you would set the following property:

Properties

```
spring.devtools.restart.exclude=static/**,public/**
```

Yaml

```
spring:
  devtools:
    restart:
      exclude: "static/**,public/**"
```

TIP If you want to keep those defaults and *add* additional exclusions, use the `spring.devtools.restart.additional-exclude` property instead.

Watching Additional Paths

You may want your application to be restarted or reloaded when you make changes to files that are not on the classpath. To do so, use the `spring.devtools.restart.additional-paths` property to configure additional paths to watch for changes. You can use the `spring.devtools.restart.exclude` property described earlier to control whether changes beneath the additional paths trigger a full restart or a live reload.

Disabling Restart

If you do not want to use the restart feature, you can disable it by using the `spring.devtools.restart.enabled` property. In most cases, you can set this property in your `application.properties` (doing so still initializes the restart classloader, but it does not watch for file changes).

If you need to *completely* disable restart support (for example, because it does not work with a specific library), you need to set the `spring.devtools.restart.enabled System` property to `false` before calling `SpringApplication.run(…)`, as shown in the following example:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        System.setProperty("spring.devtools.restart.enabled", "false");
        SpringApplication.run(MyApplication.class, args);
    }

}
```

Kotlin

```
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication

@SpringBootApplication
object MyApplication {

    @JvmStatic
    fun main(args: Array<String>) {
        System.setProperty("spring.devtools.restart.enabled", "false")
        SpringApplication.run(MyApplication::class.java, *args)
    }

}
```

Using a Trigger File

If you work with an IDE that continuously compiles changed files, you might prefer to trigger restarts only at specific times. To do so, you can use a "trigger file", which is a special file that must be modified when you want to actually trigger a restart check.

NOTE	Any update to the file will trigger a check, but restart only actually occurs if Devtools has detected it has something to do.

To use a trigger file, set the `spring.devtools.restart.trigger-file` property to the name (excluding any path) of your trigger file. The trigger file must appear somewhere on your classpath.

For example, if you have a project with the following structure:

```
src
+- main
   +- resources
      +- .reloadtrigger
```

Then your `trigger-file` property would be:

Properties

```
spring.devtools.restart.trigger-file=.reloadtrigger
```

Yaml

```
spring:
  devtools:
    restart:
      trigger-file: ".reloadtrigger"
```

Restarts will now only happen when the `src/main/resources/.reloadtrigger` is updated.

TIP	You might want to set `spring.devtools.restart.trigger-file` as a global setting, so that all your projects behave in the same way.

Some IDEs have features that save you from needing to update your trigger file manually. Spring Tools for Eclipse and IntelliJ IDEA (Ultimate Edition) both have such support. With Spring Tools, you can use the "reload" button from the console view (as long as your `trigger-file` is named `.reloadtrigger`). For IntelliJ IDEA, you can follow the instructions in their documentation.

Customizing the Restart Classloader

As described earlier in the Restart vs Reload section, restart functionality is implemented by using two classloaders. If this causes issues, you might need to customize what gets loaded by which classloader.

By default, any open project in your IDE is loaded with the "restart" classloader, and any regular `.jar` file is loaded with the "base" classloader. The same is true if you use `mvn spring-boot:run` or `gradle bootRun`: the project containing your `@SpringBootApplication` is loaded with the "restart" classloader, and everything else with the "base" classloader.

You can instruct Spring Boot to load parts of your project with a different classloader by creating a `META-INF/spring-devtools.properties` file. The `spring-devtools.properties` file can contain properties prefixed with `restart.exclude` and `restart.include`. The `include` elements are items that should be pulled up into the "restart" classloader, and the `exclude` elements are items that should be pushed down into the "base" classloader. The value of the property is a regex pattern that is applied to the classpath, as shown in the following example:

Properties

```
restart.exclude.companycommonlibs=/mycorp-common-[\\w\\d-\\.]+\\.jar
restart.include.projectcommon=/mycorp-myproj-[\\w\\d-\\.]+\\.jar
```

Yaml

```
restart:
  exclude:
    companycommonlibs: "/mycorp-common-[\\w\\d-\\.]+\\.jar"
  include:
    projectcommon: "/mycorp-myproj-[\\w\\d-\\.]+\\.jar"
```

NOTE	All property keys must be unique. As long as a property starts with `restart.include.` or `restart.exclude.` it is considered.

TIP	All `META-INF/spring-devtools.properties` from the classpath are loaded. You can package files inside your project, or in the libraries that the project consumes.

Known Limitations

Restart functionality does not work well with objects that are deserialized by using a standard `ObjectInputStream`. If you need to deserialize data, you may need to use Spring's `ConfigurableObjectInputStream` in combination with `Thread.currentThread().getContextClassLoader()`.

Unfortunately, several third-party libraries deserialize without considering the context classloader. If you find such a problem, you need to request a fix with the original authors.

6.8.4. LiveReload

The `spring-boot-devtools` module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. LiveReload browser extensions are freely available for Chrome, Firefox and Safari. You can find these extensions by searching 'LiveReload' in the marketplace or store of your chosen browser.

If you do not want to start the LiveReload server when your application runs, you can set the `spring.devtools.livereload.enabled` property to `false`.

NOTE	You can only run one LiveReload server at a time. Before starting your application, ensure that no other LiveReload servers are running. If you start multiple applications from your IDE, only the first has LiveReload support.
WARNING	To trigger LiveReload when a file changes, Automatic Restart must be enabled.

6.8.5. Global Settings

You can configure global devtools settings by adding any of the following files to the `$HOME/.config/spring-boot` directory:

1. `spring-boot-devtools.properties`
2. `spring-boot-devtools.yaml`
3. `spring-boot-devtools.yml`

Any properties added to these files apply to *all* Spring Boot applications on your machine that use devtools. For example, to configure restart to always use a trigger file, you would add the following property to your `spring-boot-devtools` file:

Properties

```
spring.devtools.restart.trigger-file=.reloadtrigger
```

Yaml

```
spring:
  devtools:
    restart:
      trigger-file: ".reloadtrigger"
```

By default, `$HOME` is the user's home directory. To customize this location, set the `SPRING_DEVTOOLS_HOME` environment variable or the `spring.devtools.home` system property.

NOTE	If devtools configuration files are not found in `$HOME/.config/spring-boot`, the root of the `$HOME` directory is searched for the presence of a `.spring-boot-devtools.properties` file. This allows you to share the devtools global configuration with applications that are on an older version of Spring Boot that does not support the `$HOME/.config/spring-boot` location.

NOTE	Profiles are not supported in devtools properties/yaml files. Any profiles activated in `.spring-boot-devtools.properties` will not affect the loading of profile-specific configuration files. Profile specific filenames (of the form `spring-boot-devtools-<profile>.properties`) and `spring.config.activate.on-profile` documents in both YAML and Properties files are not supported.

Configuring File System Watcher

FileSystemWatcher works by polling the class changes with a certain time interval, and then waiting for a predefined quiet period to make sure there are no more changes. Since Spring Boot relies entirely on the IDE to compile and copy files into the location from where Spring Boot can read them, you might find that there are times when certain changes are not reflected when devtools restarts the application. If you observe such problems constantly, try increasing the `spring.devtools.restart.poll-interval` and `spring.devtools.restart.quiet-period` parameters to the values that fit your development environment:

Properties

```
spring.devtools.restart.poll-interval=2s
spring.devtools.restart.quiet-period=1s
```

Yaml

```
spring:
  devtools:
    restart:
      poll-interval: "2s"
      quiet-period: "1s"
```

The monitored classpath directories are now polled every 2 seconds for changes, and a 1 second quiet period is maintained to make sure there are no additional class changes.

6.8.6. Remote Applications

The Spring Boot developer tools are not limited to local development. You can also use several features when running applications remotely. Remote support is opt-in as enabling it can be a security risk. It should only be enabled when running on a trusted network or when secured with SSL. If neither of these options is available to you, you should not use DevTools' remote support. You should never enable support on a production deployment.

To enable it, you need to make sure that `devtools` is included in the repackaged archive, as shown in the following listing:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludeDevtools>false</excludeDevtools>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Then you need to set the `spring.devtools.remote.secret` property. Like any important password or secret, the value should be unique and strong such that it cannot be guessed or brute-forced.

Remote devtools support is provided in two parts: a server-side endpoint that accepts connections and a client application that you run in your IDE. The server component is automatically enabled when the `spring.devtools.remote.secret` property is set. The client component must be launched manually.

NOTE	Remote devtools is not supported for Spring WebFlux applications.

Running the Remote Client Application

The remote client application is designed to be run from within your IDE. You need to run `org.springframework.boot.devtools.RemoteSpringApplication` with the same classpath as the remote project that you connect to. The application's single required argument is the remote URL to which it connects.

For example, if you are using Eclipse or Spring Tools and you have a project named `my-app` that you have deployed to Cloud Foundry, you would do the following:

- Select `Run Configurations…` from the `Run` menu.
- Create a new `Java Application` "launch configuration".
- Browse for the `my-app` project.
- Use `org.springframework.boot.devtools.RemoteSpringApplication` as the main class.

- Add `https://myapp.cfapps.io` to the `Program arguments` (or whatever your remote URL is).

A running remote client might resemble the following listing:

```
  .   ____          _                                              __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _          ___               _      \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` |        | _ \___ _ __  ___| |_ ___ \ \ \ \
 \\/  ___)| |_)| | | | | || (_| []::::::[]   / -_) '  \/ _ \  _/ -_) ) ) ) )
  '  |____| .__|_| |_|_| |_\__, |        |_|_\___|_|_|_\___/\__\___|/ / / /
 =========|_|==============|___/=================================/_/_/_/
 :: Spring Boot Remote ::  (v3.1.6)

2023-11-23T11:08:32.082Z  INFO 36896 --- [           main]
o.s.b.devtools.RemoteSpringApplication   : Starting RemoteSpringApplication v3.1.6
using Java 17.0.9 with PID 36896
(/Users/myuser/.m2/repository/org/springframework/boot/spring-boot-
devtools/3.1.6/spring-boot-devtools-3.1.6.jar started by myuser in /opt/apps/)
2023-11-23T11:08:32.088Z  INFO 36896 --- [           main]
o.s.b.devtools.RemoteSpringApplication   : No active profile set, falling back to 1
default profile: "default"
2023-11-23T11:08:32.551Z  INFO 36896 --- [           main]
o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2023-11-23T11:08:32.587Z  INFO 36896 --- [           main]
o.s.b.devtools.RemoteSpringApplication   : Started RemoteSpringApplication in 1.022
seconds (process running for 1.401)
```

NOTE	Because the remote client is using the same classpath as the real application it can directly read application properties. This is how the `spring.devtools.remote.secret` property is read and passed to the server for authentication.

TIP	It is always advisable to use `https://` as the connection protocol, so that traffic is encrypted and passwords cannot be intercepted.

TIP	If you need to use a proxy to access the remote application, configure the `spring.devtools.remote.proxy.host` and `spring.devtools.remote.proxy.port` properties.

Remote Update

The remote client monitors your application classpath for changes in the same way as the local restart. Any updated resource is pushed to the remote application and (*if required*) triggers a restart. This can be helpful if you iterate on a feature that uses a cloud service that you do not have locally. Generally, remote updates and restarts are much quicker than a full rebuild and deploy cycle.

On a slower development environment, it may happen that the quiet period is not enough, and the changes in the classes may be split into batches. The server is restarted after the first batch of class changes is uploaded. The next batch can't be sent to the application, since the server is restarting.

This is typically manifested by a warning in the `RemoteSpringApplication` logs about failing to upload some of the classes, and a consequent retry. But it may also lead to application code inconsistency and failure to restart after the first batch of changes is uploaded. If you observe such problems constantly, try increasing the `spring.devtools.restart.poll-interval` and `spring.devtools.restart.quiet-period` parameters to the values that fit your development environment. See the Configuring File System Watcher section for configuring these properties.

NOTE	Files are only monitored when the remote client is running. If you change a file before starting the remote client, it is not pushed to the remote server.

6.9. Packaging Your Application for Production

Executable jars can be used for production deployment. As they are self-contained, they are also ideally suited for cloud-based deployment.

For additional "production ready" features, such as health, auditing, and metric REST or JMX end-points, consider adding `spring-boot-actuator`. See *Production-ready Features* for details.

6.10. What to Read Next

You should now understand how you can use Spring Boot and some best practices that you should follow. You can now go on to learn about specific *Spring Boot features* in depth, or you could skip ahead and read about the "production ready" aspects of Spring Boot.

Chapter 7. Core Features

This section dives into the details of Spring Boot. Here you can learn about the key features that you may want to use and customize. If you have not already done so, you might want to read the "Getting Started" and "Developing with Spring Boot" sections, so that you have a good grounding of the basics.

7.1. SpringApplication

The `SpringApplication` class provides a convenient way to bootstrap a Spring application that is started from a `main()` method. In many situations, you can delegate to the static `SpringApplication.run` method, as shown in the following example:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication


@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

When your application starts, you should see something similar to the following output:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.1.6)

2023-11-23T11:08:33.521Z  INFO 37019 --- [           main]
o.s.b.d.f.logexample.MyApplication       : Starting MyApplication using Java 17.0.9
with PID 37019 (/opt/apps/myapp.jar started by myuser in /opt/apps/)
2023-11-23T11:08:33.527Z  INFO 37019 --- [           main]
o.s.b.d.f.logexample.MyApplication       : No active profile set, falling back to 1
default profile: "default"
2023-11-23T11:08:35.130Z  INFO 37019 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080
(http)
2023-11-23T11:08:35.152Z  INFO 37019 --- [           main]
o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-11-23T11:08:35.153Z  INFO 37019 --- [           main]
o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache
Tomcat/10.1.16]
2023-11-23T11:08:35.321Z  INFO 37019 --- [           main]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded
WebApplicationContext
2023-11-23T11:08:35.323Z  INFO 37019 --- [           main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization
completed in 1718 ms
2023-11-23T11:08:35.822Z  INFO 37019 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with
context path ''
2023-11-23T11:08:35.834Z  INFO 37019 --- [           main]
o.s.b.d.f.logexample.MyApplication       : Started MyApplication in 2.826 seconds
(process running for 3.225)
```

By default, `INFO` logging messages are shown, including some relevant startup details, such as the user that launched the application. If you need a log level other than `INFO`, you can set it, as described in Log Levels. The application version is determined using the implementation version from the main application class's package. Startup information logging can be turned off by setting `spring.main.log-startup-info` to `false`. This will also turn off logging of the application's active profiles.

TIP To add additional logging during startup, you can override `logStartupInfo(boolean)` in a subclass of `SpringApplication`.

7.1.1. Startup Failure

If your application fails to start, registered `FailureAnalyzers` get a chance to provide a dedicated error message and a concrete action to fix the problem. For instance, if you start a web application

on port `8080` and that port is already in use, you should see something similar to the following message:

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Embedded servlet container failed to start. Port 8080 was already in use.

Action:

Identify and stop the process that is listening on port 8080 or configure this
application to listen on another port.
```

NOTE	Spring Boot provides numerous `FailureAnalyzer` implementations, and you can add your own.

If no failure analyzers are able to handle the exception, you can still display the full conditions report to better understand what went wrong. To do so, you need to enable the `debug` property or enable `DEBUG` logging for `org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener`.

For instance, if you are running your application by using `java -jar`, you can enable the `debug` property as follows:

```
$ java -jar myproject-0.0.1-SNAPSHOT.jar --debug
```

7.1.2. Lazy Initialization

`SpringApplication` allows an application to be initialized lazily. When lazy initialization is enabled, beans are created as they are needed rather than during application startup. As a result, enabling lazy initialization can reduce the time that it takes your application to start. In a web application, enabling lazy initialization will result in many web-related beans not being initialized until an HTTP request is received.

A downside of lazy initialization is that it can delay the discovery of a problem with the application. If a misconfigured bean is initialized lazily, a failure will no longer occur during startup and the problem will only become apparent when the bean is initialized. Care must also be taken to ensure that the JVM has sufficient memory to accommodate all of the application's beans and not just those that are initialized during startup. For these reasons, lazy initialization is not enabled by default and it is recommended that fine-tuning of the JVM's heap size is done before enabling lazy initialization.

Lazy initialization can be enabled programmatically using the `lazyInitialization` method on `SpringApplicationBuilder` or the `setLazyInitialization` method on `SpringApplication`. Alternatively,

it can be enabled using the `spring.main.lazy-initialization` property as shown in the following example:

Properties

```
spring.main.lazy-initialization=true
```

Yaml

```
spring:
  main:
    lazy-initialization: true
```

TIP If you want to disable lazy initialization for certain beans while using lazy initialization for the rest of the application, you can explicitly set their lazy attribute to false using the `@Lazy(false)` annotation.

7.1.3. Customizing the Banner

The banner that is printed on start up can be changed by adding a `banner.txt` file to your classpath or by setting the `spring.banner.location` property to the location of such a file. If the file has an encoding other than UTF-8, you can set `spring.banner.charset`.

Inside your `banner.txt` file, you can use any key available in the `Environment` as well as any of the following placeholders:

Table 4. Banner variables

Variable	Description
`${application.version}`	The version number of your application, as declared in `MANIFEST.MF`. For example, `Implementation-Version: 1.0` is printed as `1.0`.
`${application.formatted-version}`	The version number of your application, as declared in `MANIFEST.MF` and formatted for display (surrounded with brackets and prefixed with `v`). For example `(v1.0)`.
`${spring-boot.version}`	The Spring Boot version that you are using. For example `3.1.6`.
`${spring-boot.formatted-version}`	The Spring Boot version that you are using, formatted for display (surrounded with brackets and prefixed with `v`). For example `(v3.1.6)`.
`${Ansi.NAME}` (or `${AnsiColor.NAME}`, `${AnsiBackground.NAME}`, `${AnsiStyle.NAME}`)	Where `NAME` is the name of an ANSI escape code. See `AnsiPropertySource` for details.
`${application.title}`	The title of your application, as declared in `MANIFEST.MF`. For example `Implementation-Title: MyApp` is printed as `MyApp`.

TIP	The `SpringApplication.setBanner(…)` method can be used if you want to generate a banner programmatically. Use the `org.springframework.boot.Banner` interface and implement your own `printBanner()` method.

You can also use the `spring.main.banner-mode` property to determine if the banner has to be printed on `System.out` (`console`), sent to the configured logger (`log`), or not produced at all (`off`).

The printed banner is registered as a singleton bean under the following name: `springBootBanner`.

NOTE	The `application.title`, `application.version`, and `application.formatted-version` properties are only available if you are using `java -jar` or `java -cp` with Spring Boot launchers. The values will not be resolved if you are running an unpacked jar and starting it with `java -cp <classpath> <mainclass>` or running your application as a native image. To use the `application.` properties, launch your application as a packed jar using `java -jar` or as an unpacked jar using `java org.springframework.boot.loader.JarLauncher`. This will initialize the `application.` banner properties before building the classpath and launching your app.

7.1.4. Customizing SpringApplication

If the `SpringApplication` defaults are not to your taste, you can instead create a local instance and customize it. For example, to turn off the banner, you could write:

Java

```
import org.springframework.boot.Banner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(MyApplication.class);
        application.setBannerMode(Banner.Mode.OFF);
        application.run(args);
    }

}
```

Kotlin

```
import org.springframework.boot.Banner
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args) {
        setBannerMode(Banner.Mode.OFF)
    }
}
```

NOTE	The constructor arguments passed to `SpringApplication` are configuration sources for Spring beans. In most cases, these are references to `@Configuration` classes, but they could also be direct references `@Component` classes.

It is also possible to configure the `SpringApplication` by using an `application.properties` file. See *Externalized Configuration* for details.

For a complete list of the configuration options, see the `SpringApplication` Javadoc.

7.1.5. Fluent Builder API

If you need to build an `ApplicationContext` hierarchy (multiple contexts with a parent/child relationship) or if you prefer using a "fluent" builder API, you can use the `SpringApplicationBuilder`.

The `SpringApplicationBuilder` lets you chain together multiple method calls and includes `parent` and `child` methods that let you create a hierarchy, as shown in the following example:

Java

```
new SpringApplicationBuilder().sources(Parent.class)
    .child(Application.class)
    .bannerMode(Banner.Mode.OFF)
    .run(args);
```

Kotlin

```
SpringApplicationBuilder()
    .sources(Parent::class.java)
    .child(Application::class.java)
    .bannerMode(Banner.Mode.OFF)
    .run(*args)
```

NOTE	There are some restrictions when creating an `ApplicationContext` hierarchy. For example, Web components **must** be contained within the child context, and the same `Environment` is used for both parent and child contexts. See the `SpringApplicationBuilder` Javadoc for full details.

7.1.6. Application Availability

When deployed on platforms, applications can provide information about their availability to the platform using infrastructure such as Kubernetes Probes. Spring Boot includes out-of-the box support for the commonly used "liveness" and "readiness" availability states. If you are using Spring Boot's "actuator" support then these states are exposed as health endpoint groups.

In addition, you can also obtain availability states by injecting the `ApplicationAvailability` interface into your own beans.

Liveness State

The "Liveness" state of an application tells whether its internal state allows it to work correctly, or recover by itself if it is currently failing. A broken "Liveness" state means that the application is in a state that it cannot recover from, and the infrastructure should restart the application.

NOTE	In general, the "Liveness" state should not be based on external checks, such as Health checks. If it did, a failing external system (a database, a Web API, an external cache) would trigger massive restarts and cascading failures across the platform.

The internal state of Spring Boot applications is mostly represented by the Spring `ApplicationContext`. If the application context has started successfully, Spring Boot assumes that the application is in a valid state. An application is considered live as soon as the context has been refreshed, see Spring Boot application lifecycle and related Application Events.

Readiness State

The "Readiness" state of an application tells whether the application is ready to handle traffic. A failing "Readiness" state tells the platform that it should not route traffic to the application for now. This typically happens during startup, while `CommandLineRunner` and `ApplicationRunner` components are being processed, or at any time if the application decides that it is too busy for additional traffic.

An application is considered ready as soon as application and command-line runners have been called, see Spring Boot application lifecycle and related Application Events.

TIP	Tasks expected to run during startup should be executed by `CommandLineRunner` and `ApplicationRunner` components instead of using Spring component lifecycle callbacks such as `@PostConstruct`.

Managing the Application Availability State

Application components can retrieve the current availability state at any time, by injecting the `ApplicationAvailability` interface and calling methods on it. More often, applications will want to

listen to state updates or update the state of the application.

For example, we can export the "Readiness" state of the application to a file so that a Kubernetes "exec Probe" can look at this file:

Java

```
import org.springframework.boot.availability.AvailabilityChangeEvent;
import org.springframework.boot.availability.ReadinessState;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
public class MyReadinessStateExporter {

    @EventListener
    public void onStateChange(AvailabilityChangeEvent<ReadinessState> event) {
        switch (event.getState()) {
            case ACCEPTING_TRAFFIC -> {
                // create file /tmp/healthy
            }
            case REFUSING_TRAFFIC -> {
                // remove file /tmp/healthy
            }
        }
    }

}
```

Kotlin

```
import org.springframework.boot.availability.AvailabilityChangeEvent
import org.springframework.boot.availability.ReadinessState
import org.springframework.context.event.EventListener
import org.springframework.stereotype.Component

@Component
class MyReadinessStateExporter {

    @EventListener
    fun onStateChange(event: AvailabilityChangeEvent<ReadinessState?>) {
        when (event.state) {
            ReadinessState.ACCEPTING_TRAFFIC -> {
                // create file /tmp/healthy
            }
            ReadinessState.REFUSING_TRAFFIC -> {
                // remove file /tmp/healthy
            }
            else -> {
                // ...
            }
        }
    }

}
```

We can also update the state of the application, when the application breaks and cannot recover:

Java

```
import org.springframework.boot.availability.AvailabilityChangeEvent;
import org.springframework.boot.availability.LivenessState;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.stereotype.Component;

@Component
public class MyLocalCacheVerifier {

    private final ApplicationEventPublisher eventPublisher;

    public MyLocalCacheVerifier(ApplicationEventPublisher eventPublisher) {
        this.eventPublisher = eventPublisher;
    }

    public void checkLocalCache() {
        try {
            // ...
        }
        catch (CacheCompletelyBrokenException ex) {
            AvailabilityChangeEvent.publish(this.eventPublisher, ex,
LivenessState.BROKEN);
        }
    }

}
```

Kotlin

```
import org.springframework.boot.availability.AvailabilityChangeEvent
import org.springframework.boot.availability.LivenessState
import org.springframework.context.ApplicationEventPublisher
import org.springframework.stereotype.Component

@Component
class MyLocalCacheVerifier(private val eventPublisher: ApplicationEventPublisher) {

    fun checkLocalCache() {
        try {
            // ...
        } catch (ex: CacheCompletelyBrokenException) {
            AvailabilityChangeEvent.publish(eventPublisher, ex, LivenessState.BROKEN)
        }
    }

}
```

Spring Boot provides Kubernetes HTTP probes for "Liveness" and "Readiness" with Actuator Health Endpoints. You can get more guidance about deploying Spring Boot applications on Kubernetes in

the dedicated section.

7.1.7. Application Events and Listeners

In addition to the usual Spring Framework events, such as `ContextRefreshedEvent`, a `SpringApplication` sends some additional application events.

NOTE

Some events are actually triggered before the `ApplicationContext` is created, so you cannot register a listener on those as a `@Bean`. You can register them with the `SpringApplication.addListeners(…)` method or the `SpringApplicationBuilder.listeners(…)` method.

If you want those listeners to be registered automatically, regardless of the way the application is created, you can add a `META-INF/spring.factories` file to your project and reference your listener(s) by using the `org.springframework.context.ApplicationListener` key, as shown in the following example:

```
org.springframework.context.ApplicationListener=com.example.project.MyListener
```

Application events are sent in the following order, as your application runs:

1. An `ApplicationStartingEvent` is sent at the start of a run but before any processing, except for the registration of listeners and initializers.
2. An `ApplicationEnvironmentPreparedEvent` is sent when the `Environment` to be used in the context is known but before the context is created.
3. An `ApplicationContextInitializedEvent` is sent when the `ApplicationContext` is prepared and ApplicationContextInitializers have been called but before any bean definitions are loaded.
4. An `ApplicationPreparedEvent` is sent just before the refresh is started but after bean definitions have been loaded.
5. An `ApplicationStartedEvent` is sent after the context has been refreshed but before any application and command-line runners have been called.
6. An `AvailabilityChangeEvent` is sent right after with `LivenessState.CORRECT` to indicate that the application is considered as live.
7. An `ApplicationReadyEvent` is sent after any application and command-line runners have been called.
8. An `AvailabilityChangeEvent` is sent right after with `ReadinessState.ACCEPTING_TRAFFIC` to indicate that the application is ready to service requests.
9. An `ApplicationFailedEvent` is sent if there is an exception on startup.

The above list only includes `SpringApplicationEvent`s that are tied to a `SpringApplication`. In addition to these, the following events are also published after `ApplicationPreparedEvent` and before `ApplicationStartedEvent`:

- A `WebServerInitializedEvent` is sent after the `WebServer` is ready. `ServletWebServerInitializedEvent` and `ReactiveWebServerInitializedEvent` are the servlet and reactive variants respectively.
- A `ContextRefreshedEvent` is sent when an `ApplicationContext` is refreshed.

TIP	You often need not use application events, but it can be handy to know that they exist. Internally, Spring Boot uses events to handle a variety of tasks.

NOTE	Event listeners should not run potentially lengthy tasks as they execute in the same thread by default. Consider using application and command-line runners instead.

Application events are sent by using Spring Framework's event publishing mechanism. Part of this mechanism ensures that an event published to the listeners in a child context is also published to the listeners in any ancestor contexts. As a result of this, if your application uses a hierarchy of `SpringApplication` instances, a listener may receive multiple instances of the same type of application event.

To allow your listener to distinguish between an event for its context and an event for a descendant context, it should request that its application context is injected and then compare the injected context with the context of the event. The context can be injected by implementing `ApplicationContextAware` or, if the listener is a bean, by using `@Autowired`.

7.1.8. Web Environment

A `SpringApplication` attempts to create the right type of `ApplicationContext` on your behalf. The algorithm used to determine a `WebApplicationType` is the following:

- If Spring MVC is present, an `AnnotationConfigServletWebServerApplicationContext` is used
- If Spring MVC is not present and Spring WebFlux is present, an `AnnotationConfigReactiveWebServerApplicationContext` is used
- Otherwise, `AnnotationConfigApplicationContext` is used

This means that if you are using Spring MVC and the new `WebClient` from Spring WebFlux in the same application, Spring MVC will be used by default. You can override that easily by calling `setWebApplicationType(WebApplicationType)`.

It is also possible to take complete control of the `ApplicationContext` type that is used by calling `setApplicationContextFactory(…)`.

TIP	It is often desirable to call `setWebApplicationType(WebApplicationType.NONE)` when using `SpringApplication` within a JUnit test.

7.1.9. Accessing Application Arguments

If you need to access the application arguments that were passed to `SpringApplication.run(…)`, you can inject a `org.springframework.boot.ApplicationArguments` bean. The `ApplicationArguments` interface provides access to both the raw `String[]` arguments as well as parsed `option` and `non-`

`option` arguments, as shown in the following example:

Java

```
import java.util.List;

import org.springframework.boot.ApplicationArguments;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    public MyBean(ApplicationArguments args) {
        boolean debug = args.containsOption("debug");
        List<String> files = args.getNonOptionArgs();
        if (debug) {
            System.out.println(files);
        }
        // if run with "--debug logfile.txt" prints ["logfile.txt"]
    }

}
```

Kotlin

```
import org.springframework.boot.ApplicationArguments
import org.springframework.stereotype.Component

@Component
class MyBean(args: ApplicationArguments) {

    init {
        val debug = args.containsOption("debug")
        val files = args.nonOptionArgs
        if (debug) {
            println(files)
        }
        // if run with "--debug logfile.txt" prints ["logfile.txt"]
    }

}
```

TIP Spring Boot also registers a `CommandLinePropertySource` with the Spring `Environment`. This lets you also inject single application arguments by using the `@Value` annotation.

7.1.10. Using the ApplicationRunner or CommandLineRunner

If you need to run some specific code once the `SpringApplication` has started, you can implement the `ApplicationRunner` or `CommandLineRunner` interfaces. Both interfaces work in the same way and

offer a single `run` method, which is called just before `SpringApplication.run(…)` completes.

NOTE	This contract is well suited for tasks that should run after application startup but before it starts accepting traffic.

The `CommandLineRunner` interfaces provides access to application arguments as a string array, whereas the `ApplicationRunner` uses the `ApplicationArguments` interface discussed earlier. The following example shows a `CommandLineRunner` with a `run` method:

Java

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class MyCommandLineRunner implements CommandLineRunner {

    @Override
    public void run(String... args) {
        // Do something...
    }

}
```

Kotlin

```
import org.springframework.boot.CommandLineRunner
import org.springframework.stereotype.Component

@Component
class MyCommandLineRunner : CommandLineRunner {

    override fun run(vararg args: String) {
        // Do something...
    }

}
```

If several `CommandLineRunner` or `ApplicationRunner` beans are defined that must be called in a specific order, you can additionally implement the `org.springframework.core.Ordered` interface or use the `org.springframework.core.annotation.Order` annotation.

7.1.11. Application Exit

Each `SpringApplication` registers a shutdown hook with the JVM to ensure that the `ApplicationContext` closes gracefully on exit. All the standard Spring lifecycle callbacks (such as the `DisposableBean` interface or the `@PreDestroy` annotation) can be used.

In addition, beans may implement the `org.springframework.boot.ExitCodeGenerator` interface if they

wish to return a specific exit code when `SpringApplication.exit()` is called. This exit code can then be passed to `System.exit()` to return it as a status code, as shown in the following example:

Java

```
import org.springframework.boot.ExitCodeGenerator;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class MyApplication {

    @Bean
    public ExitCodeGenerator exitCodeGenerator() {
        return () -> 42;
    }

    public static void main(String[] args) {
        System.exit(SpringApplication.exit(SpringApplication.run(MyApplication.class,
args)));
    }

}
```

Kotlin

```
import org.springframework.boot.ExitCodeGenerator
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.context.annotation.Bean

import kotlin.system.exitProcess

@SpringBootApplication
class MyApplication {

    @Bean
    fun exitCodeGenerator() = ExitCodeGenerator { 42 }

}

fun main(args: Array<String>) {
    exitProcess(SpringApplication.exit(
        runApplication<MyApplication>(*args)))
}
```

Also, the `ExitCodeGenerator` interface may be implemented by exceptions. When such an exception is encountered, Spring Boot returns the exit code provided by the implemented `getExitCode()`

method.

If there is more than one `ExitCodeGenerator`, the first non-zero exit code that is generated is used. To control the order in which the generators are called, additionally implement the `org.springframework.core.Ordered` interface or use the `org.springframework.core.annotation.Order` annotation.

7.1.12. Admin Features

It is possible to enable admin-related features for the application by specifying the `spring.application.admin.enabled` property. This exposes the `SpringApplicationAdminMXBean` on the platform `MBeanServer`. You could use this feature to administer your Spring Boot application remotely. This feature could also be useful for any service wrapper implementation.

TIP	If you want to know on which HTTP port the application is running, get the property with a key of `local.server.port`.

7.1.13. Application Startup tracking

During the application startup, the `SpringApplication` and the `ApplicationContext` perform many tasks related to the application lifecycle, the beans lifecycle or even processing application events. With `ApplicationStartup`, Spring Framework allows you to track the application startup sequence with `StartupStep` objects. This data can be collected for profiling purposes, or just to have a better understanding of an application startup process.

You can choose an `ApplicationStartup` implementation when setting up the `SpringApplication` instance. For example, to use the `BufferingApplicationStartup`, you could write:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.metrics.buffering.BufferingApplicationStartup;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(MyApplication.class);
        application.setApplicationStartup(new BufferingApplicationStartup(2048));
        application.run(args);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.context.metrics.buffering.BufferingApplicationStartup
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args) {
        applicationStartup = BufferingApplicationStartup(2048)
    }
}
```

The first available implementation, `FlightRecorderApplicationStartup` is provided by Spring Framework. It adds Spring-specific startup events to a Java Flight Recorder session and is meant for profiling applications and correlating their Spring context lifecycle with JVM events (such as allocations, GCs, class loading…). Once configured, you can record data by running the application with the Flight Recorder enabled:

```
$ java -XX:StartFlightRecording:filename=recording.jfr,duration=10s -jar demo.jar
```

Spring Boot ships with the `BufferingApplicationStartup` variant; this implementation is meant for buffering the startup steps and draining them into an external metrics system. Applications can ask for the bean of type `BufferingApplicationStartup` in any component.

Spring Boot can also be configured to expose a `startup` endpoint that provides this information as a JSON document.

7.2. Externalized Configuration

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use a variety of external configuration sources including Java properties files, YAML files, environment variables, and command-line arguments.

Property values can be injected directly into your beans by using the `@Value` annotation, accessed through Spring's `Environment` abstraction, or be bound to structured objects through `@ConfigurationProperties`.

Spring Boot uses a very particular `PropertySource` order that is designed to allow sensible overriding of values. Later property sources can override the values defined in earlier ones. Sources are considered in the following order:

1. Default properties (specified by setting `SpringApplication.setDefaultProperties`).
2. `@PropertySource` annotations on your `@Configuration` classes. Please note that such property sources are not added to the `Environment` until the application context is being refreshed. This is

too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.

3. Config data (such as `application.properties` files).
4. A `RandomValuePropertySource` that has properties only in `random.*`.
5. OS environment variables.
6. Java System properties (`System.getProperties()`).
7. JNDI attributes from `java:comp/env`.
8. `ServletContext` init parameters.
9. `ServletConfig` init parameters.
10. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. `properties` attribute on your tests. Available on `@SpringBootTest` and the test annotations for testing a particular slice of your application.
13. `@DynamicPropertySource` annotations in your tests.
14. `@TestPropertySource` annotations on your tests.
15. Devtools global settings properties in the `$HOME/.config/spring-boot` directory when devtools is active.

Config data files are considered in the following order:

1. Application properties packaged inside your jar (`application.properties` and YAML variants).
2. Profile-specific application properties packaged inside your jar (`application-{profile}.properties` and YAML variants).
3. Application properties outside of your packaged jar (`application.properties` and YAML variants).
4. Profile-specific application properties outside of your packaged jar (`application-{profile}.properties` and YAML variants).

NOTE	It is recommended to stick with one format for your entire application. If you have configuration files with both `.properties` and YAML format in the same location, `.properties` takes precedence.
NOTE	If you use environment variables rather than system properties, most operating systems disallow period-separated key names, but you can use underscores instead (for example, `SPRING_CONFIG_NAME` instead of `spring.config.name`). See Binding From Environment Variables for details.
NOTE	If your application runs in a servlet container or application server, then JNDI properties (in `java:comp/env`) or servlet context initialization parameters can be used instead of, or as well as, environment variables or system properties.

To provide a concrete example, suppose you develop a `@Component` that uses a `name` property, as shown in the following example:

Java

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @Value("${name}")
    private String name;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Value
import org.springframework.stereotype.Component

@Component
class MyBean {

    @Value("\${name}")
    private val name: String? = null

    // ...

}
```

On your application classpath (for example, inside your jar) you can have an `application.properties` file that provides a sensible default property value for `name`. When running in a new environment, an `application.properties` file can be provided outside of your jar that overrides the `name`. For one-off testing, you can launch with a specific command line switch (for example, `java -jar app.jar --name="Spring"`).

TIP The `env` and `configprops` endpoints can be useful in determining why a property has a particular value. You can use these two endpoints to diagnose unexpected property values. See the "Production ready features" section for details.

7.2.1. Accessing Command Line Properties

By default, `SpringApplication` converts any command line option arguments (that is, arguments starting with `--`, such as `--server.port=9000`) to a `property` and adds them to the Spring `Environment`. As mentioned previously, command line properties always take precedence over file-based property sources.

If you do not want command line properties to be added to the `Environment`, you can disable them by using `SpringApplication.setAddCommandLineProperties(false)`.

7.2.2. JSON Application Properties

Environment variables and system properties often have restrictions that mean some property names cannot be used. To help with this, Spring Boot allows you to encode a block of properties into a single JSON structure.

When your application starts, any `spring.application.json` or `SPRING_APPLICATION_JSON` properties will be parsed and added to the `Environment`.

For example, the `SPRING_APPLICATION_JSON` property can be supplied on the command line in a UN*X shell as an environment variable:

```
$ SPRING_APPLICATION_JSON='{"my":{"name":"test"}}' java -jar myapp.jar
```

In the preceding example, you end up with `my.name=test` in the Spring `Environment`.

The same JSON can also be provided as a system property:

```
$ java -Dspring.application.json='{"my":{"name":"test"}}' -jar myapp.jar
```

Or you could supply the JSON by using a command line argument:

```
$ java -jar myapp.jar --spring.application.json='{"my":{"name":"test"}}'
```

If you are deploying to a classic Application Server, you could also use a JNDI variable named `java:comp/env/spring.application.json`.

NOTE	Although `null` values from the JSON will be added to the resulting property source, the `PropertySourcesPropertyResolver` treats `null` properties as missing values. This means that the JSON cannot override properties from lower order property sources with a `null` value.

7.2.3. External Application Properties

Spring Boot will automatically find and load `application.properties` and `application.yaml` files from the following locations when your application starts:

1. From the classpath
    a. The classpath root
    b. The classpath `/config` package
2. From the current directory

a. The current directory

b. The `config/` subdirectory in the current directory

c. Immediate child directories of the `config/` subdirectory

The list is ordered by precedence (with values from lower items overriding earlier ones). Documents from the loaded files are added as `PropertySources` to the Spring `Environment`.

If you do not like `application` as the configuration file name, you can switch to another file name by specifying a `spring.config.name` environment property. For example, to look for `myproject.properties` and `myproject.yaml` files you can run your application as follows:

```
$ java -jar myproject.jar --spring.config.name=myproject
```

You can also refer to an explicit location by using the `spring.config.location` environment property. This property accepts a comma-separated list of one or more locations to check.

The following example shows how to specify two distinct files:

```
$ java -jar myproject.jar --spring.config.location=\
    optional:classpath:/default.properties,\
    optional:classpath:/override.properties
```

TIP Use the prefix `optional:` if the locations are optional and you do not mind if they do not exist.

WARNING `spring.config.name`, `spring.config.location`, and `spring.config.additional-location` are used very early to determine which files have to be loaded. They must be defined as an environment property (typically an OS environment variable, a system property, or a command-line argument).

If `spring.config.location` contains directories (as opposed to files), they should end in `/`. At runtime they will be appended with the names generated from `spring.config.name` before being loaded. Files specified in `spring.config.location` are imported directly.

NOTE Both directory and file location values are also expanded to check for profile-specific files. For example, if you have a `spring.config.location` of `classpath:myconfig.properties`, you will also find appropriate `classpath:myconfig-<profile>.properties` files are loaded.

In most situations, each `spring.config.location` item you add will reference a single file or directory. Locations are processed in the order that they are defined and later ones can override the values of earlier ones.

If you have a complex location setup, and you use profile-specific configuration files, you may need to provide further hints so that Spring Boot knows how they should be grouped. A location group is a collection of locations that are all considered at the same level. For example, you might want to

group all classpath locations, then all external locations. Items within a location group should be separated with `;`. See the example in the "Profile Specific Files" section for more details.

Locations configured by using `spring.config.location` replace the default locations. For example, if `spring.config.location` is configured with the value `optional:classpath:/custom-config/,optional:file:./custom-config/`, the complete set of locations considered is:

1. `optional:classpath:custom-config/`
2. `optional:file:./custom-config/`

If you prefer to add additional locations, rather than replacing them, you can use `spring.config.additional-location`. Properties loaded from additional locations can override those in the default locations. For example, if `spring.config.additional-location` is configured with the value `optional:classpath:/custom-config/,optional:file:./custom-config/`, the complete set of locations considered is:

1. `optional:classpath:/;optional:classpath:/config/`
2. `optional:file:./;optional:file:./config/;optional:file:./config/*/`
3. `optional:classpath:custom-config/`
4. `optional:file:./custom-config/`

This search ordering lets you specify default values in one configuration file and then selectively override those values in another. You can provide default values for your application in `application.properties` (or whatever other basename you choose with `spring.config.name`) in one of the default locations. These default values can then be overridden at runtime with a different file located in one of the custom locations.

Optional Locations

By default, when a specified config data location does not exist, Spring Boot will throw a `ConfigDataLocationNotFoundException` and your application will not start.

If you want to specify a location, but you do not mind if it does not always exist, you can use the `optional:` prefix. You can use this prefix with the `spring.config.location` and `spring.config.additional-location` properties, as well as with `spring.config.import` declarations.

For example, a `spring.config.import` value of `optional:file:./myconfig.properties` allows your application to start, even if the `myconfig.properties` file is missing.

If you want to ignore all `ConfigDataLocationNotFoundExceptions` and always continue to start your application, you can use the `spring.config.on-not-found` property. Set the value to `ignore` using `SpringApplication.setDefaultProperties(…)` or with a system/environment variable.

Wildcard Locations

If a config file location includes the `*` character for the last path segment, it is considered a wildcard location. Wildcards are expanded when the config is loaded so that immediate subdirectories are also checked. Wildcard locations are particularly useful in an environment such as Kubernetes when there are multiple sources of config properties.

For example, if you have some Redis configuration and some MySQL configuration, you might want to keep those two pieces of configuration separate, while requiring that both those are present in an `application.properties` file. This might result in two separate `application.properties` files mounted at different locations such as `/config/redis/application.properties` and `/config/mysql/application.properties`. In such a case, having a wildcard location of `config/*/`, will result in both files being processed.

By default, Spring Boot includes `config/*/` in the default search locations. It means that all subdirectories of the `/config` directory outside of your jar will be searched.

You can use wildcard locations yourself with the `spring.config.location` and `spring.config.additional-location` properties.

NOTE	A wildcard location must contain only one `*` and end with `*/` for search locations that are directories or `*/<filename>` for search locations that are files. Locations with wildcards are sorted alphabetically based on the absolute path of the file names.

TIP	Wildcard locations only work with external directories. You cannot use a wildcard in a `classpath:` location.

Profile Specific Files

As well as `application` property files, Spring Boot will also attempt to load profile-specific files using the naming convention `application-{profile}`. For example, if your application activates a profile named `prod` and uses YAML files, then both `application.yaml` and `application-prod.yaml` will be considered.

Profile-specific properties are loaded from the same locations as standard `application.properties`, with profile-specific files always overriding the non-specific ones. If several profiles are specified, a last-wins strategy applies. For example, if profiles `prod,live` are specified by the `spring.profiles.active` property, values in `application-prod.properties` can be overridden by those in `application-live.properties`.

The last-wins strategy applies at the location group level. A `spring.config.location` of `classpath:/cfg/,classpath:/ext/` will not have the same override rules as `classpath:/cfg/;classpath:/ext/`.

For example, continuing our `prod,live` example above, we might have the following files:

```
/cfg
  application-live.properties
/ext
  application-live.properties
  application-prod.properties
```

NOTE

When we have a `spring.config.location` of `classpath:/cfg/,classpath:/ext/` we process all `/cfg` files before all `/ext` files:

1. `/cfg/application-live.properties`
2. `/ext/application-prod.properties`
3. `/ext/application-live.properties`

When we have `classpath:/cfg/;classpath:/ext/` instead (with a `;` delimiter) we process `/cfg` and `/ext` at the same level:

1. `/ext/application-prod.properties`
2. `/cfg/application-live.properties`
3. `/ext/application-live.properties`

The `Environment` has a set of default profiles (by default, `[default]`) that are used if no active profiles are set. In other words, if no profiles are explicitly activated, then properties from `application-default` are considered.

NOTE

Properties files are only ever loaded once. If you have already directly imported a profile specific property files then it will not be imported a second time.

Importing Additional Data

Application properties may import further config data from other locations using the `spring.config.import` property. Imports are processed as they are discovered, and are treated as additional documents inserted immediately below the one that declares the import.

For example, you might have the following in your classpath `application.properties` file:

Properties

```
spring.application.name=myapp
spring.config.import=optional:file:./dev.properties
```

Yaml

```
spring:
  application:
    name: "myapp"
  config:
    import: "optional:file:./dev.properties"
```

This will trigger the import of a `dev.properties` file in current directory (if such a file exists). Values from the imported `dev.properties` will take precedence over the file that triggered the import. In the above example, the `dev.properties` could redefine `spring.application.name` to a different value.

An import will only be imported once no matter how many times it is declared. The order an import is defined inside a single document within the properties/yaml file does not matter. For instance, the two examples below produce the same result:

Properties

```
spring.config.import=my.properties
my.property=value
```

Yaml

```
spring:
  config:
    import: "my.properties"
my:
  property: "value"
```

Properties

```
my.property=value
spring.config.import=my.properties
```

Yaml

```
my:
  property: "value"
spring:
  config:
    import: "my.properties"
```

In both of the above examples, the values from the `my.properties` file will take precedence over the file that triggered its import.

Several locations can be specified under a single `spring.config.import` key. Locations will be processed in the order that they are defined, with later imports taking precedence.

NOTE	When appropriate, Profile-specific variants are also considered for import. The example above would import both `my.properties` as well as any `my-<profile>.properties` variants.

TIP	Spring Boot includes pluggable API that allows various different location addresses to be supported. By default you can import Java Properties, YAML and "configuration trees". Third-party jars can offer support for additional technologies (there is no requirement for files to be local). For example, you can imagine config data being from external stores such as Consul, Apache ZooKeeper or Netflix Archaius. If you want to support your own locations, see the `ConfigDataLocationResolver` and `ConfigDataLoader` classes in the `org.springframework.boot.context.config` package.

Importing Extensionless Files

Some cloud platforms cannot add a file extension to volume mounted files. To import these extensionless files, you need to give Spring Boot a hint so that it knows how to load them. You can do this by putting an extension hint in square brackets.

For example, suppose you have a `/etc/config/myconfig` file that you wish to import as yaml. You can import it from your `application.properties` using the following:

Properties

```
spring.config.import=file:/etc/config/myconfig[.yaml]
```

Yaml

```
spring:
  config:
    import: "file:/etc/config/myconfig[.yaml]"
```

Using Configuration Trees

When running applications on a cloud platform (such as Kubernetes) you often need to read config values that the platform supplies. It is not uncommon to use environment variables for such purposes, but this can have drawbacks, especially if the value is supposed to be kept secret.

As an alternative to environment variables, many cloud platforms now allow you to map configuration into mounted data volumes. For example, Kubernetes can volume mount both `ConfigMaps` and `Secrets`.

There are two common volume mount patterns that can be used:

1. A single file contains a complete set of properties (usually written as YAML).
2. Multiple files are written to a directory tree, with the filename becoming the 'key' and the

contents becoming the 'value'.

For the first case, you can import the YAML or Properties file directly using `spring.config.import` as described above. For the second case, you need to use the `configtree:` prefix so that Spring Boot knows it needs to expose all the files as properties.

As an example, let's imagine that Kubernetes has mounted the following volume:

```
etc/
  config/
    myapp/
      username
      password
```

The contents of the `username` file would be a config value, and the contents of `password` would be a secret.

To import these properties, you can add the following to your `application.properties` or `application.yaml` file:

Properties

```
spring.config.import=optional:configtree:/etc/config/
```

Yaml

```
spring:
  config:
    import: "optional:configtree:/etc/config/"
```

You can then access or inject `myapp.username` and `myapp.password` properties from the `Environment` in the usual way.

TIP The names of the folders and files under the config tree form the property name. In the above example, to access the properties as `username` and `password`, you can set `spring.config.import` to `optional:configtree:/etc/config/myapp`.

NOTE Filenames with dot notation are also correctly mapped. For example, in the above example, a file named `myapp.username` in `/etc/config` would result in a `myapp.username` property in the `Environment`.

TIP Configuration tree values can be bound to both string `String` and `byte[]` types depending on the contents expected.

If you have multiple config trees to import from the same parent folder you can use a wildcard shortcut. Any `configtree:` location that ends with `/*/` will import all immediate children as config trees. As with a non-wildcard import, the names of the folders and files under each config tree form

the property name.

For example, given the following volume:

```
etc/
  config/
    dbconfig/
      db/
        username
        password
    mqconfig/
      mq/
        username
        password
```

You can use `configtree:/etc/config/*/` as the import location:

Properties

```
spring.config.import=optional:configtree:/etc/config/*/
```

Yaml

```
spring:
  config:
    import: "optional:configtree:/etc/config/*/"
```

This will add `db.username`, `db.password`, `mq.username` and `mq.password` properties.

NOTE	Directories loaded using a wildcard are sorted alphabetically. If you need a different order, then you should list each location as a separate import

Configuration trees can also be used for Docker secrets. When a Docker swarm service is granted access to a secret, the secret gets mounted into the container. For example, if a secret named `db.password` is mounted at location `/run/secrets/`, you can make `db.password` available to the Spring environment using the following:

Properties

```
spring.config.import=optional:configtree:/run/secrets/
```

Yaml

```
spring:
  config:
    import: "optional:configtree:/run/secrets/"
```

Property Placeholders

The values in `application.properties` and `application.yaml` are filtered through the existing `Environment` when they are used, so you can refer back to previously defined values (for example, from System properties or environment variables). The standard `${name}` property-placeholder syntax can be used anywhere within a value. Property placeholders can also specify a default value using a `:` to separate the default value from the property name, for example `${name:default}`.

The use of placeholders with and without defaults is shown in the following example:

Properties

```
app.name=MyApp
app.description=${app.name} is a Spring Boot application written by
${username:Unknown}
```

Yaml

```
app:
  name: "MyApp"
  description: "${app.name} is a Spring Boot application written by
${username:Unknown}"
```

Assuming that the `username` property has not been set elsewhere, `app.description` will have the value `MyApp is a Spring Boot application written by Unknown`.

NOTE

You should always refer to property names in the placeholder using their canonical form (kebab-case using only lowercase letters). This will allow Spring Boot to use the same logic as it does when relaxed binding `@ConfigurationProperties`.

For example, `${demo.item-price}` will pick up `demo.item-price` and `demo.itemPrice` forms from the `application.properties` file, as well as `DEMO_ITEMPRICE` from the system environment. If you used `${demo.itemPrice}` instead, `demo.item-price` and `DEMO_ITEMPRICE` would not be considered.

TIP

You can also use this technique to create "short" variants of existing Spring Boot properties. See the *Use 'Short' Command Line Arguments* how-to for details.

Working With Multi-Document Files

Spring Boot allows you to split a single physical file into multiple logical documents which are each added independently. Documents are processed in order, from top to bottom. Later documents can override the properties defined in earlier ones.

For `application.yaml` files, the standard YAML multi-document syntax is used. Three consecutive hyphens represent the end of one document, and the start of the next.

For example, the following file has two logical documents:

```
spring:
  application:
    name: "MyApp"
---
spring:
  application:
    name: "MyCloudApp"
  config:
    activate:
      on-cloud-platform: "kubernetes"
```

For `application.properties` files a special `#---` or `!---` comment is used to mark the document splits:

```
spring.application.name=MyApp
#---
spring.application.name=MyCloudApp
spring.config.activate.on-cloud-platform=kubernetes
```

NOTE	Property file separators must not have any leading whitespace and must have exactly three hyphen characters. The lines immediately before and after the separator must not be same comment prefix.

TIP	Multi-document property files are often used in conjunction with activation properties such as `spring.config.activate.on-profile`. See the next section for details.

WARNING	Multi-document property files cannot be loaded by using the `@PropertySource` or `@TestPropertySource` annotations.

Activation Properties

It is sometimes useful to only activate a given set of properties when certain conditions are met. For example, you might have properties that are only relevant when a specific profile is active.

You can conditionally activate a properties document using `spring.config.activate.*`.

The following activation properties are available:

Table 5. activation properties

Property	Note
`on-profile`	A profile expression that must match for the document to be active.
`on-cloud-platform`	The `CloudPlatform` that must be detected for the document to be active.

For example, the following specifies that the second document is only active when running on Kubernetes, and only when either the “prod” or “staging” profiles are active:

Properties

```
myprop=always-set
#---
spring.config.activate.on-cloud-platform=kubernetes
spring.config.activate.on-profile=prod | staging
myotherprop=sometimes-set
```

Yaml

```
myprop:
  "always-set"
---
spring:
  config:
    activate:
      on-cloud-platform: "kubernetes"
      on-profile: "prod | staging"
myotherprop: "sometimes-set"
```

7.2.4. Encrypting Properties

Spring Boot does not provide any built-in support for encrypting property values, however, it does provide the hook points necessary to modify values contained in the Spring `Environment`. The `EnvironmentPostProcessor` interface allows you to manipulate the `Environment` before the application starts. See Customize the Environment or ApplicationContext Before It Starts for details.

If you need a secure way to store credentials and passwords, the Spring Cloud Vault project provides support for storing externalized configuration in HashiCorp Vault.

7.2.5. Working With YAML

YAML is a superset of JSON and, as such, is a convenient format for specifying hierarchical configuration data. The `SpringApplication` class automatically supports YAML as an alternative to properties whenever you have the SnakeYAML library on your classpath.

NOTE	If you use “Starters”, SnakeYAML is automatically provided by `spring-boot-starter`.

Mapping YAML to Properties

YAML documents need to be converted from their hierarchical format to a flat structure that can be used with the Spring `Environment`. For example, consider the following YAML document:

```
environments:
  dev:
    url: "https://dev.example.com"
    name: "Developer Setup"
  prod:
    url: "https://another.example.com"
    name: "My Cool App"
```

In order to access these properties from the `Environment`, they would be flattened as follows:

```
environments.dev.url=https://dev.example.com
environments.dev.name=Developer Setup
environments.prod.url=https://another.example.com
environments.prod.name=My Cool App
```

Likewise, YAML lists also need to be flattened. They are represented as property keys with `[index]` dereferencers. For example, consider the following YAML:

```
my:
 servers:
 - "dev.example.com"
 - "another.example.com"
```

The preceding example would be transformed into these properties:

```
my.servers[0]=dev.example.com
my.servers[1]=another.example.com
```

TIP	Properties that use the `[index]` notation can be bound to Java `List` or `Set` objects using Spring Boot's `Binder` class. For more details see the "Type-safe Configuration Properties" section below.

WARNING	YAML files cannot be loaded by using the `@PropertySource` or `@TestPropertySource` annotations. So, in the case that you need to load values that way, you need to use a properties file.

Directly Loading YAML

Spring Framework provides two convenient classes that can be used to load YAML documents. The `YamlPropertiesFactoryBean` loads YAML as `Properties` and the `YamlMapFactoryBean` loads YAML as a `Map`.

You can also use the `YamlPropertySourceLoader` class if you want to load YAML as a Spring `PropertySource`.

7.2.6. Configuring Random Values

The `RandomValuePropertySource` is useful for injecting random values (for example, into secrets or test cases). It can produce integers, longs, uuids, or strings, as shown in the following example:

Properties

```
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.uuid=${random.uuid}
my.number-less-than-ten=${random.int(10)}
my.number-in-range=${random.int[1024,65536]}
```

Yaml

```
my:
  secret: "${random.value}"
  number: "${random.int}"
  bignumber: "${random.long}"
  uuid: "${random.uuid}"
  number-less-than-ten: "${random.int(10)}"
  number-in-range: "${random.int[1024,65536]}"
```

The `random.int*` syntax is `OPEN value (,max) CLOSE` where the `OPEN,CLOSE` are any character and `value,max` are integers. If `max` is provided, then `value` is the minimum value and `max` is the maximum value (exclusive).

7.2.7. Configuring System Environment Properties

Spring Boot supports setting a prefix for environment properties. This is useful if the system environment is shared by multiple Spring Boot applications with different configuration requirements. The prefix for system environment properties can be set directly on `SpringApplication`.

For example, if you set the prefix to `input`, a property such as `remote.timeout` will also be resolved as `input.remote.timeout` in the system environment.

7.2.8. Type-safe Configuration Properties

Using the `@Value("${property}")` annotation to inject configuration properties can sometimes be cumbersome, especially if you are working with multiple properties or your data is hierarchical in nature. Spring Boot provides an alternative method of working with properties that lets strongly typed beans govern and validate the configuration of your application.

TIP See also the differences between `@Value` and type-safe configuration properties.

JavaBean Properties Binding

It is possible to bind a bean declaring standard JavaBean properties as shown in the following example:

Java

```
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("my.service")
public class MyProperties {

    private boolean enabled;

    private InetAddress remoteAddress;

    private final Security security = new Security();

    public boolean isEnabled() {
        return this.enabled;
    }

    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

    public InetAddress getRemoteAddress() {
        return this.remoteAddress;
    }

    public void setRemoteAddress(InetAddress remoteAddress) {
        this.remoteAddress = remoteAddress;
    }

    public Security getSecurity() {
        return this.security;
    }

    public static class Security {

        private String username;

        private String password;

        private List<String> roles = new ArrayList<>(Collections.singleton("USER"));
```

```
        public String getUsername() {
            return this.username;
        }

        public void setUsername(String username) {
            this.username = username;
        }

        public String getPassword() {
            return this.password;
        }

        public void setPassword(String password) {
            this.password = password;
        }

        public List<String> getRoles() {
            return this.roles;
        }

        public void setRoles(List<String> roles) {
            this.roles = roles;
        }

    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import java.net.InetAddress

@ConfigurationProperties("my.service")
class MyProperties {

    var isEnabled = false

    var remoteAddress: InetAddress? = null

    val security = Security()

    class Security {

        var username: String? = null

        var password: String? = null

        var roles: List<String> = ArrayList(setOf("USER"))

    }

}
```

The preceding POJO defines the following properties:

- `my.service.enabled`, with a value of `false` by default.
- `my.service.remote-address`, with a type that can be coerced from `String`.
- `my.service.security.username`, with a nested "security" object whose name is determined by the name of the property. In particular, the type is not used at all there and could have been `SecurityProperties`.
- `my.service.security.password`.
- `my.service.security.roles`, with a collection of `String` that defaults to `USER`.

NOTE The properties that map to `@ConfigurationProperties` classes available in Spring Boot, which are configured through properties files, YAML files, environment variables, and other mechanisms, are public API but the accessors (getters/setters) of the class itself are not meant to be used directly.

NOTE

Such arrangement relies on a default empty constructor and getters and setters are usually mandatory, since binding is through standard Java Beans property descriptors, just like in Spring MVC. A setter may be omitted in the following cases:

- Maps, as long as they are initialized, need a getter but not necessarily a setter, since they can be mutated by the binder.
- Collections and arrays can be accessed either through an index (typically with YAML) or by using a single comma-separated value (properties). In the latter case, a setter is mandatory. We recommend to always add a setter for such types. If you initialize a collection, make sure it is not immutable (as in the preceding example).
- If nested POJO properties are initialized (like the `Security` field in the preceding example), a setter is not required. If you want the binder to create the instance on the fly by using its default constructor, you need a setter.

Some people use Project Lombok to add getters and setters automatically. Make sure that Lombok does not generate any particular constructor for such a type, as it is used automatically by the container to instantiate the object.

Finally, only standard Java Bean properties are considered and binding on static properties is not supported.

Constructor Binding

The example in the previous section can be rewritten in an immutable fashion as shown in the following example:

Java

```
import java.net.InetAddress;
import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.bind.DefaultValue;

@ConfigurationProperties("my.service")
public class MyProperties {

    private final boolean enabled;

    private final InetAddress remoteAddress;

    private final Security security;


    public MyProperties(boolean enabled, InetAddress remoteAddress, Security security)
{
        this.enabled = enabled;
        this.remoteAddress = remoteAddress;
```

```
        this.security = security;
    }

    public boolean isEnabled() {
        return this.enabled;
    }

    public InetAddress getRemoteAddress() {
        return this.remoteAddress;
    }

    public Security getSecurity() {
        return this.security;
    }

    public static class Security {

        private final String username;

        private final String password;

        private final List<String> roles;


        public Security(String username, String password, @DefaultValue("USER")
List<String> roles) {
            this.username = username;
            this.password = password;
            this.roles = roles;
        }

        public String getUsername() {
            return this.username;
        }

        public String getPassword() {
            return this.password;
        }

        public List<String> getRoles() {
            return this.roles;
        }

    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.context.properties.bind.DefaultValue
import java.net.InetAddress

@ConfigurationProperties("my.service")
class MyProperties(val enabled: Boolean, val remoteAddress: InetAddress,
        val security: Security) {

    class Security(val username: String, val password: String,
            @param:DefaultValue("USER") val roles: List<String>)

}
```

In this setup, the presence of a single parameterized constructor implies that constructor binding should be used. This means that the binder will find a constructor with the parameters that you wish to have bound. If your class has multiple constructors, the `@ConstructorBinding` annotation can be used to specify which constructor to use for constructor binding. To opt out of constructor binding for a class with a single parameterized constructor, the constructor must be annotated with `@Autowired`. Constructor binding can be used with records. Unless your record has multiple constructors, there is no need to use `@ConstructorBinding`.

Nested members of a constructor bound class (such as `Security` in the example above) will also be bound through their constructor.

Default values can be specified using `@DefaultValue` on constructor parameters and record components. The conversion service will be applied to coerce the annotation's `String` value to the target type of a missing property.

Referring to the previous example, if no properties are bound to `Security`, the `MyProperties` instance will contain a `null` value for `security`. To make it contain a non-null instance of `Security` even when no properties are bound to it (when using Kotlin, this will require the `username` and `password` parameters of `Security` to be declared as nullable as they do not have default values), use an empty `@DefaultValue` annotation:

Java

```
public MyProperties(boolean enabled, InetAddress remoteAddress, @DefaultValue Security
security) {
    this.enabled = enabled;
    this.remoteAddress = remoteAddress;
    this.security = security;
}
```

Kotlin

```
class MyProperties(val enabled: Boolean, val remoteAddress: InetAddress,
        @DefaultValue val security: Security) {

    class Security(val username: String?, val password: String?,
            @param:DefaultValue("USER") val roles: List<String>)

}
```

NOTE	To use constructor binding the class must be enabled using `@EnableConfigurationProperties` or configuration property scanning. You cannot use constructor binding with beans that are created by the regular Spring mechanisms (for example `@Component` beans, beans created by using `@Bean` methods or beans loaded by using `@Import`)
NOTE	To use constructor binding in a native image the class must be compiled with `-parameters`. This will happen automatically if you use Spring Boot's Gradle plugin or if you use Maven and `spring-boot-starter-parent`.
NOTE	The use of `java.util.Optional` with `@ConfigurationProperties` is not recommended as it is primarily intended for use as a return type. As such, it is not well-suited to configuration property injection. For consistency with properties of other types, if you do declare an `Optional` property and it has no value, `null` rather than an empty `Optional` will be bound.

Enabling @ConfigurationProperties-annotated Types

Spring Boot provides infrastructure to bind `@ConfigurationProperties` types and register them as beans. You can either enable configuration properties on a class-by-class basis or enable configuration property scanning that works in a similar manner to component scanning.

Sometimes, classes annotated with `@ConfigurationProperties` might not be suitable for scanning, for example, if you're developing your own auto-configuration or you want to enable them conditionally. In these cases, specify the list of types to process using the `@EnableConfigurationProperties` annotation. This can be done on any `@Configuration` class, as shown in the following example:

Java

```
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(SomeProperties.class)
public class MyConfiguration {

}
```

Kotlin

```
import org.springframework.boot.context.properties.EnableConfigurationProperties
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(SomeProperties::class)
class MyConfiguration
```

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("some.properties")
public class SomeProperties {

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties

@ConfigurationProperties("some.properties")
class SomeProperties
```

To use configuration property scanning, add the `@ConfigurationPropertiesScan` annotation to your application. Typically, it is added to the main application class that is annotated with `@SpringBootApplication` but it can be added to any `@Configuration` class. By default, scanning will occur from the package of the class that declares the annotation. If you want to define specific packages to scan, you can do so as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.ConfigurationPropertiesScan;

@SpringBootApplication
@ConfigurationPropertiesScan({ "com.example.app", "com.example.another" })
public class MyApplication {

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.context.properties.ConfigurationPropertiesScan

@SpringBootApplication
@ConfigurationPropertiesScan("com.example.app", "com.example.another")
class MyApplication
```

NOTE

When the `@ConfigurationProperties` bean is registered using configuration property scanning or through `@EnableConfigurationProperties`, the bean has a conventional name: `<prefix>-<fqn>`, where `<prefix>` is the environment key prefix specified in the `@ConfigurationProperties` annotation and `<fqn>` is the fully qualified name of the bean. If the annotation does not provide any prefix, only the fully qualified name of the bean is used.

Assuming that it is in the `com.example.app` package, the bean name of the `SomeProperties` example above is `some.properties-com.example.app.SomeProperties`.

We recommend that `@ConfigurationProperties` only deal with the environment and, in particular, does not inject other beans from the context. For corner cases, setter injection can be used or any of the `*Aware` interfaces provided by the framework (such as `EnvironmentAware` if you need access to the `Environment`). If you still want to inject other beans using the constructor, the configuration properties bean must be annotated with `@Component` and use JavaBean-based property binding.

Using @ConfigurationProperties-annotated Types

This style of configuration works particularly well with the `SpringApplication` external YAML configuration, as shown in the following example:

```
my:
  service:
    remote-address: 192.168.1.1
    security:
      username: "admin"
      roles:
      - "USER"
      - "ADMIN"
```

To work with `@ConfigurationProperties` beans, you can inject them in the same way as any other bean, as shown in the following example:

Java

```
import org.springframework.stereotype.Service;

@Service
public class MyService {

    private final MyProperties properties;

    public MyService(MyProperties properties) {
        this.properties = properties;
    }

    public void openConnection() {
        Server server = new Server(this.properties.getRemoteAddress());
        server.start();
        // ...
    }

    // ...

}
```

Kotlin

```
import org.springframework.stereotype.Service

@Service
class MyService(val properties: MyProperties) {

    fun openConnection() {
        val server = Server(properties.remoteAddress)
        server.start()
        // ...
    }

    // ...

}
```

TIP Using `@ConfigurationProperties` also lets you generate metadata files that can be used by IDEs to offer auto-completion for your own keys. See the appendix for details.

Third-party Configuration

As well as using `@ConfigurationProperties` to annotate a class, you can also use it on public `@Bean` methods. Doing so can be particularly useful when you want to bind properties to third-party components that are outside of your control.

To configure a bean from the `Environment` properties, add `@ConfigurationProperties` to its bean registration, as shown in the following example:

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class ThirdPartyConfiguration {

    @Bean
    @ConfigurationProperties(prefix = "another")
    public AnotherComponent anotherComponent() {
        return new AnotherComponent();
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class ThirdPartyConfiguration {

    @Bean
    @ConfigurationProperties(prefix = "another")
    fun anotherComponent(): AnotherComponent = AnotherComponent()

}
```

Any JavaBean property defined with the `another` prefix is mapped onto that `AnotherComponent` bean in manner similar to the preceding `SomeProperties` example.

Relaxed Binding

Spring Boot uses some relaxed rules for binding `Environment` properties to `@ConfigurationProperties` beans, so there does not need to be an exact match between the `Environment` property name and the bean property name. Common examples where this is useful include dash-separated environment properties (for example, `context-path` binds to `contextPath`), and capitalized environment properties (for example, `PORT` binds to `port`).

As an example, consider the following `@ConfigurationProperties` class:

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "my.main-project.person")
public class MyPersonProperties {

    private String firstName;

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties

@ConfigurationProperties(prefix = "my.main-project.person")
class MyPersonProperties {

    var firstName: String? = null

}
```

With the preceding code, the following properties names can all be used:

Table 6. relaxed binding

Property	Note
`my.main-project.person.first-name`	Kebab case, which is recommended for use in `.properties` and YAML files.
`my.main-project.person.firstName`	Standard camel case syntax.
`my.main-project.person.first_name`	Underscore notation, which is an alternative format for use in `.properties` and YAML files.
`MY_MAINPROJECT_PERSON_FIRSTNAME`	Upper case format, which is recommended when using system environment variables.

NOTE The `prefix` value for the annotation *must* be in kebab case (lowercase and separated by `-`, such as `my.main-project.person`).

Table 7. relaxed binding rules per property source

Property Source	Simple	List
Properties Files	Camel case, kebab case, or underscore notation	Standard list syntax using `[ ]` or comma-separated values
YAML Files	Camel case, kebab case, or underscore notation	Standard YAML list syntax or comma-separated values
Environment Variables	Upper case format with underscore as the delimiter (see Binding From Environment Variables).	Numeric values surrounded by underscores (see Binding From Environment Variables)
System properties	Camel case, kebab case, or underscore notation	Standard list syntax using `[ ]` or comma-separated values

TIP We recommend that, when possible, properties are stored in lower-case kebab format, such as `my.person.first-name=Rod`.

Binding Maps

When binding to `Map` properties you may need to use a special bracket notation so that the original `key` value is preserved. If the key is not surrounded by `[]`, any characters that are not alpha-numeric, `-` or `.` are removed.

For example, consider binding the following properties to a `Map<String,String>`:

Properties

```
my.map.[/key1]=value1
my.map.[/key2]=value2
my.map./key3=value3
```

Yaml

```
my:
  map:
    "[/key1]": "value1"
    "[/key2]": "value2"
    "/key3": "value3"
```

NOTE	For YAML files, the brackets need to be surrounded by quotes for the keys to be parsed properly.

The properties above will bind to a `Map` with `/key1`, `/key2` and `key3` as the keys in the map. The slash has been removed from `key3` because it was not surrounded by square brackets.

When binding to scalar values, keys with `.` in them do not need to be surrounded by `[]`. Scalar values include enums and all types in the `java.lang` package except for `Object`. Binding `a.b=c` to `Map<String, String>` will preserve the `.` in the key and return a Map with the entry `{"a.b"="c"}`. For any other types you need to use the bracket notation if your `key` contains a `.`. For example, binding `a.b=c` to `Map<String, Object>` will return a Map with the entry `{"a"={"b"="c"}}` whereas `[a.b]=c` will return a Map with the entry `{"a.b"="c"}`.

Binding From Environment Variables

Most operating systems impose strict rules around the names that can be used for environment variables. For example, Linux shell variables can contain only letters (`a` to `z` or `A` to `Z`), numbers (`0` to `9`) or the underscore character (`_`). By convention, Unix shell variables will also have their names in UPPERCASE.

Spring Boot's relaxed binding rules are, as much as possible, designed to be compatible with these naming restrictions.

To convert a property name in the canonical-form to an environment variable name you can follow these rules:

- Replace dots (`.`) with underscores (`_`).

- Remove any dashes (-).
- Convert to uppercase.

For example, the configuration property `spring.main.log-startup-info` would be an environment variable named `SPRING_MAIN_LOGSTARTUPINFO`.

Environment variables can also be used when binding to object lists. To bind to a `List`, the element number should be surrounded with underscores in the variable name.

For example, the configuration property `my.service[0].other` would use an environment variable named `MY_SERVICE_0_OTHER`.

Caching

Relaxed binding uses a cache to improve performance. By default, this caching is only applied to immutable property sources. To customize this behavior, for example to enable caching for mutable property sources, use `ConfigurationPropertyCaching`.

Merging Complex Types

When lists are configured in more than one place, overriding works by replacing the entire list.

For example, assume a `MyPojo` object with `name` and `description` attributes that are `null` by default. The following example exposes a list of `MyPojo` objects from `MyProperties`:

Java

```
import java.util.ArrayList;
import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("my")
public class MyProperties {

    private final List<MyPojo> list = new ArrayList<>();

    public List<MyPojo> getList() {
        return this.list;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties

@ConfigurationProperties("my")
class MyProperties {

    val list: List<MyPojo> = ArrayList()

}
```

Consider the following configuration:

Properties

```
my.list[0].name=my name
my.list[0].description=my description
#---
spring.config.activate.on-profile=dev
my.list[0].name=my another name
```

Yaml

```
my:
  list:
  - name: "my name"
    description: "my description"
---
spring:
  config:
    activate:
      on-profile: "dev"
my:
  list:
  - name: "my another name"
```

If the `dev` profile is not active, `MyProperties.list` contains one `MyPojo` entry, as previously defined. If the `dev` profile is enabled, however, the `list` *still* contains only one entry (with a name of `my another name` and a description of `null`). This configuration *does not* add a second `MyPojo` instance to the list, and it does not merge the items.

When a `List` is specified in multiple profiles, the one with the highest priority (and only that one) is used. Consider the following example:

Properties

```
my.list[0].name=my name
my.list[0].description=my description
my.list[1].name=another name
my.list[1].description=another description
#---
spring.config.activate.on-profile=dev
my.list[0].name=my another name
```

Yaml

```
my:
  list:
  - name: "my name"
    description: "my description"
  - name: "another name"
    description: "another description"
---
spring:
  config:
    activate:
      on-profile: "dev"
my:
  list:
  - name: "my another name"
```

In the preceding example, if the `dev` profile is active, `MyProperties.list` contains *one* `MyPojo` entry (with a name of `my another name` and a description of `null`). For YAML, both comma-separated lists and YAML lists can be used for completely overriding the contents of the list.

For `Map` properties, you can bind with property values drawn from multiple sources. However, for the same property in multiple sources, the one with the highest priority is used. The following example exposes a `Map<String, MyPojo>` from `MyProperties`:

Java

```
import java.util.LinkedHashMap;
import java.util.Map;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("my")
public class MyProperties {

    private final Map<String, MyPojo> map = new LinkedHashMap<>();

    public Map<String, MyPojo> getMap() {
        return this.map;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties

@ConfigurationProperties("my")
class MyProperties {

    val map: Map<String, MyPojo> = LinkedHashMap()

}
```

Consider the following configuration:

Properties

```
my.map.key1.name=my name 1
my.map.key1.description=my description 1
#---
spring.config.activate.on-profile=dev
my.map.key1.name=dev name 1
my.map.key2.name=dev name 2
my.map.key2.description=dev description 2
```

Yaml

```
my:
  map:
    key1:
      name: "my name 1"
      description: "my description 1"
---
spring:
  config:
    activate:
      on-profile: "dev"
my:
  map:
    key1:
      name: "dev name 1"
    key2:
      name: "dev name 2"
      description: "dev description 2"
```

If the `dev` profile is not active, `MyProperties.map` contains one entry with key `key1` (with a name of `my name 1` and a description of `my description 1`). If the `dev` profile is enabled, however, `map` contains two entries with keys `key1` (with a name of `dev name 1` and a description of `my description 1`) and `key2` (with a name of `dev name 2` and a description of `dev description 2`).

NOTE	The preceding merging rules apply to properties from all property sources, and not just files.

Properties Conversion

Spring Boot attempts to coerce the external application properties to the right type when it binds to the `@ConfigurationProperties` beans. If you need custom type conversion, you can provide a `ConversionService` bean (with a bean named `conversionService`) or custom property editors (through a `CustomEditorConfigurer` bean) or custom `Converters` (with bean definitions annotated as `@ConfigurationPropertiesBinding`).

NOTE	As this bean is requested very early during the application lifecycle, make sure to limit the dependencies that your `ConversionService` is using. Typically, any dependency that you require may not be fully initialized at creation time. You may want to rename your custom `ConversionService` if it is not required for configuration keys coercion and only rely on custom converters qualified with `@ConfigurationPropertiesBinding`.

Converting Durations

Spring Boot has dedicated support for expressing durations. If you expose a `java.time.Duration` property, the following formats in application properties are available:

- A regular `long` representation (using milliseconds as the default unit unless a `@DurationUnit` has

been specified)

- The standard ISO-8601 format used by `java.time.Duration`
- A more readable format where the value and the unit are coupled (`10s` means 10 seconds)

Consider the following example:

Java

```
import java.time.Duration;
import java.time.temporal.ChronoUnit;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.convert.DurationUnit;

@ConfigurationProperties("my")
public class MyProperties {

    @DurationUnit(ChronoUnit.SECONDS)
    private Duration sessionTimeout = Duration.ofSeconds(30);

    private Duration readTimeout = Duration.ofMillis(1000);

    public Duration getSessionTimeout() {
        return this.sessionTimeout;
    }

    public void setSessionTimeout(Duration sessionTimeout) {
        this.sessionTimeout = sessionTimeout;
    }

    public Duration getReadTimeout() {
        return this.readTimeout;
    }

    public void setReadTimeout(Duration readTimeout) {
        this.readTimeout = readTimeout;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.convert.DurationUnit
import java.time.Duration
import java.time.temporal.ChronoUnit

@ConfigurationProperties("my")
class MyProperties {

    @DurationUnit(ChronoUnit.SECONDS)
    var sessionTimeout = Duration.ofSeconds(30)

    var readTimeout = Duration.ofMillis(1000)

}
```

To specify a session timeout of 30 seconds, `30`, `PT30S` and `30s` are all equivalent. A read timeout of 500ms can be specified in any of the following form: `500`, `PT0.5S` and `500ms`.

You can also use any of the supported units. These are:

- `ns` for nanoseconds
- `us` for microseconds
- `ms` for milliseconds
- `s` for seconds
- `m` for minutes
- `h` for hours
- `d` for days

The default unit is milliseconds and can be overridden using `@DurationUnit` as illustrated in the sample above.

If you prefer to use constructor binding, the same properties can be exposed, as shown in the following example:

Java

```
import java.time.Duration;
import java.time.temporal.ChronoUnit;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.bind.DefaultValue;
import org.springframework.boot.convert.DurationUnit;

@ConfigurationProperties("my")
public class MyProperties {

    private final Duration sessionTimeout;

    private final Duration readTimeout;

    public MyProperties(@DurationUnit(ChronoUnit.SECONDS) @DefaultValue("30s")
Duration sessionTimeout,
            @DefaultValue("1000ms") Duration readTimeout) {
        this.sessionTimeout = sessionTimeout;
        this.readTimeout = readTimeout;
    }

    public Duration getSessionTimeout() {
        return this.sessionTimeout;
    }

    public Duration getReadTimeout() {
        return this.readTimeout;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.context.properties.bind.DefaultValue
import org.springframework.boot.convert.DurationUnit
import java.time.Duration
import java.time.temporal.ChronoUnit

@ConfigurationProperties("my")
class MyProperties(@param:DurationUnit(ChronoUnit.SECONDS) @param:DefaultValue("30s")
val sessionTimeout: Duration,
        @param:DefaultValue("1000ms") val readTimeout: Duration)
```

TIP If you are upgrading a `Long` property, make sure to define the unit (using `@DurationUnit`) if it is not milliseconds. Doing so gives a transparent upgrade path while supporting a much richer format.

Converting Periods

In addition to durations, Spring Boot can also work with `java.time.Period` type. The following formats can be used in application properties:

- An regular `int` representation (using days as the default unit unless a `@PeriodUnit` has been specified)
- The standard ISO-8601 format used by `java.time.Period`
- A simpler format where the value and the unit pairs are coupled (`1y3d` means 1 year and 3 days)

The following units are supported with the simple format:

- `y` for years
- `m` for months
- `w` for weeks
- `d` for days

NOTE	The `java.time.Period` type never actually stores the number of weeks, it is a shortcut that means "7 days".

Converting Data Sizes

Spring Framework has a `DataSize` value type that expresses a size in bytes. If you expose a `DataSize` property, the following formats in application properties are available:

- A regular `long` representation (using bytes as the default unit unless a `@DataSizeUnit` has been specified)
- A more readable format where the value and the unit are coupled (`10MB` means 10 megabytes)

Consider the following example:

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.convert.DataSizeUnit;
import org.springframework.util.unit.DataSize;
import org.springframework.util.unit.DataUnit;

@ConfigurationProperties("my")
public class MyProperties {

    @DataSizeUnit(DataUnit.MEGABYTES)
    private DataSize bufferSize = DataSize.ofMegabytes(2);

    private DataSize sizeThreshold = DataSize.ofBytes(512);

    public DataSize getBufferSize() {
        return this.bufferSize;
    }

    public void setBufferSize(DataSize bufferSize) {
        this.bufferSize = bufferSize;
    }

    public DataSize getSizeThreshold() {
        return this.sizeThreshold;
    }

    public void setSizeThreshold(DataSize sizeThreshold) {
        this.sizeThreshold = sizeThreshold;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.convert.DataSizeUnit
import org.springframework.util.unit.DataSize
import org.springframework.util.unit.DataUnit

@ConfigurationProperties("my")
class MyProperties {

    @DataSizeUnit(DataUnit.MEGABYTES)
    var bufferSize = DataSize.ofMegabytes(2)

    var sizeThreshold = DataSize.ofBytes(512)

}
```

To specify a buffer size of 10 megabytes, `10` and `10MB` are equivalent. A size threshold of 256 bytes can be specified as `256` or `256B`.

You can also use any of the supported units. These are:

- `B` for bytes
- `KB` for kilobytes
- `MB` for megabytes
- `GB` for gigabytes
- `TB` for terabytes

The default unit is bytes and can be overridden using `@DataSizeUnit` as illustrated in the sample above.

If you prefer to use constructor binding, the same properties can be exposed, as shown in the following example:

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.bind.DefaultValue;
import org.springframework.boot.convert.DataSizeUnit;
import org.springframework.util.unit.DataSize;
import org.springframework.util.unit.DataUnit;

@ConfigurationProperties("my")
public class MyProperties {

    private final DataSize bufferSize;

    private final DataSize sizeThreshold;

    public MyProperties(@DataSizeUnit(DataUnit.MEGABYTES) @DefaultValue("2MB")
DataSize bufferSize,
            @DefaultValue("512B") DataSize sizeThreshold) {
        this.bufferSize = bufferSize;
        this.sizeThreshold = sizeThreshold;
    }

    public DataSize getBufferSize() {
        return this.bufferSize;
    }

    public DataSize getSizeThreshold() {
        return this.sizeThreshold;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.context.properties.bind.DefaultValue
import org.springframework.boot.convert.DataSizeUnit
import org.springframework.util.unit.DataSize
import org.springframework.util.unit.DataUnit

@ConfigurationProperties("my")
class MyProperties(@param:DataSizeUnit(DataUnit.MEGABYTES) @param:DefaultValue("2MB")
val bufferSize: DataSize,
        @param:DefaultValue("512B") val sizeThreshold: DataSize)
```

TIP	If you are upgrading a `Long` property, make sure to define the unit (using `@DataSizeUnit`) if it is not bytes. Doing so gives a transparent upgrade path while supporting a much richer format.

@ConfigurationProperties Validation

Spring Boot attempts to validate `@ConfigurationProperties` classes whenever they are annotated with Spring's `@Validated` annotation. You can use JSR-303 `jakarta.validation` constraint annotations directly on your configuration class. To do so, ensure that a compliant JSR-303 implementation is on your classpath and then add constraint annotations to your fields, as shown in the following example:

Java

```
import java.net.InetAddress;

import jakarta.validation.constraints.NotNull;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.validation.annotation.Validated;

@ConfigurationProperties("my.service")
@Validated
public class MyProperties {

    @NotNull
    private InetAddress remoteAddress;

    public InetAddress getRemoteAddress() {
        return this.remoteAddress;
    }

    public void setRemoteAddress(InetAddress remoteAddress) {
        this.remoteAddress = remoteAddress;
    }

}
```

Kotlin

```
import jakarta.validation.constraints.NotNull
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.validation.annotation.Validated
import java.net.InetAddress

@ConfigurationProperties("my.service")
@Validated
class MyProperties {

    var remoteAddress: @NotNull InetAddress? = null

}
```

TIP You can also trigger validation by annotating the `@Bean` method that creates the configuration properties with `@Validated`.

To ensure that validation is always triggered for nested properties, even when no properties are found, the associated field must be annotated with `@Valid`. The following example builds on the preceding `MyProperties` example:

Java

```
import java.net.InetAddress;

import jakarta.validation.Valid;
import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.NotNull;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.validation.annotation.Validated;

@ConfigurationProperties("my.service")
@Validated
public class MyProperties {

    @NotNull
    private InetAddress remoteAddress;

    @Valid
    private final Security security = new Security();

    public InetAddress getRemoteAddress() {
        return this.remoteAddress;
    }

    public void setRemoteAddress(InetAddress remoteAddress) {
        this.remoteAddress = remoteAddress;
    }

    public Security getSecurity() {
        return this.security;
    }

    public static class Security {

        @NotEmpty
        private String username;

        public String getUsername() {
            return this.username;
        }

        public void setUsername(String username) {
            this.username = username;
        }

    }

}
```

Kotlin

```
import jakarta.validation.Valid
import jakarta.validation.constraints.NotEmpty
import jakarta.validation.constraints.NotNull
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.validation.annotation.Validated
import java.net.InetAddress

@ConfigurationProperties("my.service")
@Validated
class MyProperties {

    var remoteAddress: @NotNull InetAddress? = null

    @Valid
    val security = Security()

    class Security {

        @NotEmpty
        var username: String? = null

    }

}
```

You can also add a custom Spring `Validator` by creating a bean definition called `configurationPropertiesValidator`. The `@Bean` method should be declared `static`. The configuration properties validator is created very early in the application's lifecycle, and declaring the `@Bean` method as static lets the bean be created without having to instantiate the `@Configuration` class. Doing so avoids any problems that may be caused by early instantiation.

TIP The `spring-boot-actuator` module includes an endpoint that exposes all `@ConfigurationProperties` beans. Point your web browser to `/actuator/configprops` or use the equivalent JMX endpoint. See the "Production ready features" section for details.

@ConfigurationProperties vs. @Value

The `@Value` annotation is a core container feature, and it does not provide the same features as type-safe configuration properties. The following table summarizes the features that are supported by `@ConfigurationProperties` and `@Value`:

Feature	`@ConfigurationProperties`	`@Value`
Relaxed binding	Yes	Limited (see note below)

Feature	`@ConfigurationProperties`	`@Value`
Meta-data support	Yes	No
`SpEL` evaluation	No	Yes

NOTE

If you do want to use `@Value`, we recommend that you refer to property names using their canonical form (kebab-case using only lowercase letters). This will allow Spring Boot to use the same logic as it does when relaxed binding `@ConfigurationProperties`.

For example, `@Value("${demo.item-price}")` will pick up `demo.item-price` and `demo.itemPrice` forms from the `application.properties` file, as well as `DEMO_ITEMPRICE` from the system environment. If you used `@Value("${demo.itemPrice}")` instead, `demo.item-price` and `DEMO_ITEMPRICE` would not be considered.

If you define a set of configuration keys for your own components, we recommend you group them in a POJO annotated with `@ConfigurationProperties`. Doing so will provide you with structured, type-safe object that you can inject into your own beans.

`SpEL` expressions from application property files are not processed at time of parsing these files and populating the environment. However, it is possible to write a `SpEL` expression in `@Value`. If the value of a property from an application property file is a `SpEL` expression, it will be evaluated when consumed through `@Value`.

7.3. Profiles

Spring Profiles provide a way to segregate parts of your application configuration and make it be available only in certain environments. Any `@Component`, `@Configuration` or `@ConfigurationProperties` can be marked with `@Profile` to limit when it is loaded, as shown in the following example:

Java

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration(proxyBeanMethods = false)
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```

Kotlin

```
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Profile

@Configuration(proxyBeanMethods = false)
@Profile("production")
class ProductionConfiguration {

    // ...

}
```

NOTE

If `@ConfigurationProperties` beans are registered through `@EnableConfigurationProperties` instead of automatic scanning, the `@Profile` annotation needs to be specified on the `@Configuration` class that has the `@EnableConfigurationProperties` annotation. In the case where `@ConfigurationProperties` are scanned, `@Profile` can be specified on the `@ConfigurationProperties` class itself.

You can use a `spring.profiles.active` `Environment` property to specify which profiles are active. You can specify the property in any of the ways described earlier in this chapter. For example, you could include it in your `application.properties`, as shown in the following example:

Properties

```
spring.profiles.active=dev,hsqldb
```

Yaml

```
spring:
  profiles:
    active: "dev,hsqldb"
```

You could also specify it on the command line by using the following switch: `--spring.profiles.active=dev,hsqldb`.

If no profile is active, a default profile is enabled. The name of the default profile is `default` and it can be tuned using the `spring.profiles.default` `Environment` property, as shown in the following example:

Properties

```
spring.profiles.default=none
```

Yaml

```
spring:
  profiles:
    default: "none"
```

`spring.profiles.active` and `spring.profiles.default` can only be used in non-profile specific documents. This means they cannot be included in profile specific files or documents activated by `spring.config.activate.on-profile`.

For example, the second document configuration is invalid:

Properties

```
# this document is valid
spring.profiles.active=prod
#---
# this document is invalid
spring.config.activate.on-profile=prod
spring.profiles.active=metrics
```

Yaml

```
# this document is valid
spring:
  profiles:
    active: "prod"
---
# this document is invalid
spring:
  config:
    activate:
      on-profile: "prod"
  profiles:
    active: "metrics"
```

7.3.1. Adding Active Profiles

The `spring.profiles.active` property follows the same ordering rules as other properties: The highest `PropertySource` wins. This means that you can specify active profiles in `application.properties` and then **replace** them by using the command line switch.

Sometimes, it is useful to have properties that **add** to the active profiles rather than replace them. The `spring.profiles.include` property can be used to add active profiles on top of those activated by the `spring.profiles.active` property. The `SpringApplication` entry point also has a Java API for setting additional profiles. See the `setAdditionalProfiles()` method in SpringApplication.

For example, when an application with the following properties is run, the common and local profiles will be activated even when it runs using the `--spring.profiles.active` switch:

Properties

```
spring.profiles.include[0]=common
spring.profiles.include[1]=local
```

Yaml

```
spring:
  profiles:
    include:
      - "common"
      - "local"
```

WARNING	Similar to `spring.profiles.active`, `spring.profiles.include` can only be used in non-profile specific documents. This means it cannot be included in profile specific files or documents activated by `spring.config.activate.on-profile`.

Profile groups, which are described in the next section can also be used to add active profiles if a given profile is active.

7.3.2. Profile Groups

Occasionally the profiles that you define and use in your application are too fine-grained and become cumbersome to use. For example, you might have `proddb` and `prodmq` profiles that you use to enable database and messaging features independently.

To help with this, Spring Boot lets you define profile groups. A profile group allows you to define a logical name for a related group of profiles.

For example, we can create a `production` group that consists of our `proddb` and `prodmq` profiles.

Properties

```
spring.profiles.group.production[0]=proddb
spring.profiles.group.production[1]=prodmq
```

Yaml

```
spring:
  profiles:
    group:
      production:
      - "proddb"
      - "prodmq"
```

Our application can now be started using `--spring.profiles.active=production` to activate the `production`, `proddb` and `prodmq` profiles in one hit.

7.3.3. Programmatically Setting Profiles

You can programmatically set active profiles by calling `SpringApplication.setAdditionalProfiles(…)` before your application runs. It is also possible to activate profiles by using Spring's `ConfigurableEnvironment` interface.

7.3.4. Profile-specific Configuration Files

Profile-specific variants of both `application.properties` (or `application.yaml`) and files referenced through `@ConfigurationProperties` are considered as files and loaded. See "Profile Specific Files" for details.

7.4. Logging

Spring Boot uses Commons Logging for all internal logging but leaves the underlying log implementation open. Default configurations are provided for Java Util Logging, Log4j2, and Logback. In each case, loggers are pre-configured to use console output with optional file output also available.

By default, if you use the "Starters", Logback is used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J all work correctly.

TIP	There are a lot of logging frameworks available for Java. Do not worry if the above list seems confusing. Generally, you do not need to change your logging dependencies and the Spring Boot defaults work just fine.

TIP	When you deploy your application to a servlet container or application server, logging performed with the Java Util Logging API is not routed into your application's logs. This prevents logging performed by the container or other applications that have been deployed to it from appearing in your application's logs.

7.4.1. Log Format

The default log output from Spring Boot resembles the following example:

```
2023-11-23T11:08:28.856Z  INFO 36643 --- [           main]
o.s.b.d.f.logexample.MyApplication       : Starting MyApplication using Java 17.0.9
with PID 36643 (/opt/apps/myapp.jar started by myuser in /opt/apps/)
2023-11-23T11:08:28.863Z  INFO 36643 --- [           main]
o.s.b.d.f.logexample.MyApplication       : No active profile set, falling back to 1
default profile: "default"
2023-11-23T11:08:30.438Z  INFO 36643 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080
(http)
2023-11-23T11:08:30.456Z  INFO 36643 --- [           main]
o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-11-23T11:08:30.457Z  INFO 36643 --- [           main]
o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache
Tomcat/10.1.16]
2023-11-23T11:08:30.615Z  INFO 36643 --- [           main]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded
WebApplicationContext
2023-11-23T11:08:30.617Z  INFO 36643 --- [           main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization
completed in 1681 ms
2023-11-23T11:08:31.143Z  INFO 36643 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with
context path ''
2023-11-23T11:08:31.158Z  INFO 36643 --- [           main]
o.s.b.d.f.logexample.MyApplication       : Started MyApplication in 2.907 seconds
(process running for 3.322)
```

The following items are output:

- Date and Time: Millisecond precision and easily sortable.
- Log Level: `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE`.
- Process ID.
- A `---` separator to distinguish the start of actual log messages.
- Thread name: Enclosed in square brackets (may be truncated for console output).
- Logger name: This is usually the source class name (often abbreviated).
- The log message.

NOTE	Logback does not have a `FATAL` level. It is mapped to `ERROR`.

7.4.2. Console Output

The default log configuration echoes messages to the console as they are written. By default, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged. You can also enable a “debug” mode by starting your application with a `--debug` flag.

```
$ java -jar myapp.jar --debug
```

NOTE	You can also specify `debug=true` in your `application.properties`.

When the debug mode is enabled, a selection of core loggers (embedded container, Hibernate, and Spring Boot) are configured to output more information. Enabling the debug mode does *not* configure your application to log all messages with `DEBUG` level.

Alternatively, you can enable a "trace" mode by starting your application with a `--trace` flag (or `trace=true` in your `application.properties`). Doing so enables trace logging for a selection of core loggers (embedded container, Hibernate schema generation, and the whole Spring portfolio).

Color-coded Output

If your terminal supports ANSI, color output is used to aid readability. You can set `spring.output.ansi.enabled` to a supported value to override the auto-detection.

Color coding is configured by using the `%clr` conversion word. In its simplest form, the converter colors the output according to the log level, as shown in the following example:

```
%clr(%5p)
```

The following table describes the mapping of log levels to colors:

Level	Color
`FATAL`	Red
`ERROR`	Red
`WARN`	Yellow
`INFO`	Green
`DEBUG`	Green
`TRACE`	Green

Alternatively, you can specify the color or style that should be used by providing it as an option to the conversion. For example, to make the text yellow, use the following setting:

```
%clr(%d{yyyy-MM-dd'T'HH:mm:ss.SSSXXX}){yellow}
```

The following colors and styles are supported:

- `blue`
- `cyan`
- `faint`

- `green`
- `magenta`
- `red`
- `yellow`

7.4.3. File Output

By default, Spring Boot logs only to the console and does not write log files. If you want to write log files in addition to the console output, you need to set a `logging.file.name` or `logging.file.path` property (for example, in your `application.properties`).

The following table shows how the `logging.*` properties can be used together:

Table 8. Logging properties

`logging.file.name`	`logging.file.path`	**Example**	**Description**
(none)	*(none)*		Console only logging.
Specific file	*(none)*	`my.log`	Writes to the specified log file. Names can be an exact location or relative to the current directory.
(none)	Specific directory	`/var/log`	Writes `spring.log` to the specified directory. Names can be an exact location or relative to the current directory.

Log files rotate when they reach 10 MB and, as with console output, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged by default.

TIP Logging properties are independent of the actual logging infrastructure. As a result, specific configuration keys (such as `logback.configurationFile` for Logback) are not managed by spring Boot.

7.4.4. File Rotation

If you are using the Logback, it is possible to fine-tune log rotation settings using your `application.properties` or `application.yaml` file. For all other logging system, you will need to configure rotation settings directly yourself (for example, if you use Log4j2 then you could add a `log4j2.xml` or `log4j2-spring.xml` file).

The following rotation policy properties are supported:

Name	**Description**
`logging.logback.rollingpolicy.file-name-pattern`	The filename pattern used to create log archives.
`logging.logback.rollingpolicy.clean-history-on-start`	If log archive cleanup should occur when the application starts.
`logging.logback.rollingpolicy.max-file-size`	The maximum size of log file before it is archived.

Name	Description
`logging.logback.rollingpolicy.total-size-cap`	The maximum amount of size log archives can take before being deleted.
`logging.logback.rollingpolicy.max-history`	The maximum number of archive log files to keep (defaults to 7).

7.4.5. Log Levels

All the supported logging systems can have the logger levels set in the Spring `Environment` (for example, in `application.properties`) by using `logging.level.<logger-name>=<level>` where `level` is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, or OFF. The `root` logger can be configured by using `logging.level.root`.

The following example shows potential logging settings in `application.properties`:

Properties

```
logging.level.root=warn
logging.level.org.springframework.web=debug
logging.level.org.hibernate=error
```

Yaml

```
logging:
  level:
    root: "warn"
    org.springframework.web: "debug"
    org.hibernate: "error"
```

It is also possible to set logging levels using environment variables. For example, `LOGGING_LEVEL_ORG_SPRINGFRAMEWORK_WEB=DEBUG` will set `org.springframework.web` to `DEBUG`.

NOTE The above approach will only work for package level logging. Since relaxed binding always converts environment variables to lowercase, it is not possible to configure logging for an individual class in this way. If you need to configure logging for a class, you can use the `SPRING_APPLICATION_JSON` variable.

7.4.6. Log Groups

It is often useful to be able to group related loggers together so that they can all be configured at the same time. For example, you might commonly change the logging levels for *all* Tomcat related loggers, but you can not easily remember top level packages.

To help with this, Spring Boot allows you to define logging groups in your Spring `Environment`. For example, here is how you could define a “tomcat” group by adding it to your `application.properties`:

Properties

```
logging.group.tomcat=org.apache.catalina,org.apache.coyote,org.apache.tomcat
```

Yaml

```
logging:
  group:
    tomcat: "org.apache.catalina,org.apache.coyote,org.apache.tomcat"
```

Once defined, you can change the level for all the loggers in the group with a single line:

Properties

```
logging.level.tomcat=trace
```

Yaml

```
logging:
  level:
    tomcat: "trace"
```

Spring Boot includes the following pre-defined logging groups that can be used out-of-the-box:

Name	Loggers
web	`org.springframework.core.codec`, `org.springframework.http`, `org.springframework.web`, `org.springframework.boot.actuate.endpoint.web`, `org.springframework.boot.web.servlet.ServletContextInitializerBeans`
sql	`org.springframework.jdbc.core`, `org.hibernate.SQL`, `org.jooq.tools.LoggerListener`

7.4.7. Using a Log Shutdown Hook

In order to release logging resources when your application terminates, a shutdown hook that will trigger log system cleanup when the JVM exits is provided. This shutdown hook is registered automatically unless your application is deployed as a war file. If your application has complex context hierarchies the shutdown hook may not meet your needs. If it does not, disable the shutdown hook and investigate the options provided directly by the underlying logging system. For example, Logback offers context selectors which allow each Logger to be created in its own context. You can use the `logging.register-shutdown-hook` property to disable the shutdown hook. Setting it to `false` will disable the registration. You can set the property in your `application.properties` or `application.yaml` file:

Properties

```
logging.register-shutdown-hook=false
```

Yaml

```
logging:
  register-shutdown-hook: false
```

7.4.8. Custom Log Configuration

The various logging systems can be activated by including the appropriate libraries on the classpath and can be further customized by providing a suitable configuration file in the root of the classpath or in a location specified by the following Spring `Environment` property: `logging.config`.

You can force Spring Boot to use a particular logging system by using the `org.springframework.boot.logging.LoggingSystem` system property. The value should be the fully qualified class name of a `LoggingSystem` implementation. You can also disable Spring Boot's logging configuration entirely by using a value of `none`.

NOTE	Since logging is initialized **before** the `ApplicationContext` is created, it is not possible to control logging from `@PropertySources` in Spring `@Configuration` files. The only way to change the logging system or disable it entirely is through System properties.

Depending on your logging system, the following files are loaded:

Logging System	Customization
Logback	`logback-spring.xml`, `logback-spring.groovy`, `logback.xml`, or `logback.groovy`
Log4j2	`log4j2-spring.xml` or `log4j2.xml`
JDK (Java Util Logging)	`logging.properties`

NOTE	When possible, we recommend that you use the `-spring` variants for your logging configuration (for example, `logback-spring.xml` rather than `logback.xml`). If you use standard configuration locations, Spring cannot completely control log initialization.

WARNING	There are known classloading issues with Java Util Logging that cause problems when running from an 'executable jar'. We recommend that you avoid it when running from an 'executable jar' if at all possible.

To help with the customization, some other properties are transferred from the Spring `Environment` to System properties. This allows the properties to be consumed by logging system configuration. For example, setting `logging.file.name` in `application.properties` or `LOGGING_FILE_NAME` as an environment variable will result in the `LOG_FILE` System property being set. The properties that are transferred are described in the following table:

Spring Environment	System Property	Comments
logging.exception-conversion-word	LOG_EXCEPTION_CONVERSION_WORD	The conversion word used when logging exceptions.
logging.file.name	LOG_FILE	If defined, it is used in the default log configuration.
logging.file.path	LOG_PATH	If defined, it is used in the default log configuration.
logging.pattern.console	CONSOLE_LOG_PATTERN	The log pattern to use on the console (stdout).
logging.pattern.dateformat	LOG_DATEFORMAT_PATTERN	Appender pattern for log date format.
logging.charset.console	CONSOLE_LOG_CHARSET	The charset to use for console logging.
logging.threshold.console	CONSOLE_LOG_THRESHOLD	The log level threshold to use for console logging.
logging.pattern.file	FILE_LOG_PATTERN	The log pattern to use in a file (if LOG_FILE is enabled).
logging.charset.file	FILE_LOG_CHARSET	The charset to use for file logging (if LOG_FILE is enabled).
logging.threshold.file	FILE_LOG_THRESHOLD	The log level threshold to use for file logging.
logging.pattern.level	LOG_LEVEL_PATTERN	The format to use when rendering the log level (default %5p).
PID	PID	The current process ID (discovered if possible and when not already defined as an OS environment variable).

If you use Logback, the following properties are also transferred:

Spring Environment	System Property	Comments
logging.logback.rollingpolicy.file-name-pattern	LOGBACK_ROLLINGPOLICY_FILE_NAME_PATTERN	Pattern for rolled-over log file names (default ${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz).
logging.logback.rollingpolicy.clean-history-on-start	LOGBACK_ROLLINGPOLICY_CLEAN_HISTORY_ON_START	Whether to clean the archive log files on startup.
logging.logback.rollingpolicy.max-file-size	LOGBACK_ROLLINGPOLICY_MAX_FILE_SIZE	Maximum log file size.
logging.logback.rollingpolicy.total-size-cap	LOGBACK_ROLLINGPOLICY_TOTAL_SIZE_CAP	Total size of log backups to be kept.

Spring Environment	System Property	Comments
`logging.logback.rollingpolicy.max-history`	`LOGBACK_ROLLINGPOLICY_MAX_HISTORY`	Maximum number of archive log files to keep.

All the supported logging systems can consult System properties when parsing their configuration files. See the default configurations in `spring-boot.jar` for examples:

- Logback
- Log4j 2
- Java Util logging

TIP If you want to use a placeholder in a logging property, you should use Spring Boot's syntax and not the syntax of the underlying framework. Notably, if you use Logback, you should use `:` as the delimiter between a property name and its default value and not use `:-`.

TIP You can add MDC and other ad-hoc content to log lines by overriding only the `LOG_LEVEL_PATTERN` (or `logging.pattern.level` with Logback). For example, if you use `logging.pattern.level=user:%X{user} %5p`, then the default log format contains an MDC entry for "user", if it exists, as shown in the following example.

```
2019-08-30 12:30:04.031 user:someone INFO 22174 --- [  nio-8080-exec-0] demo.Controller
Handling authenticated request
```

7.4.9. Logback Extensions

Spring Boot includes a number of extensions to Logback that can help with advanced configuration. You can use these extensions in your `logback-spring.xml` configuration file.

NOTE Because the standard `logback.xml` configuration file is loaded too early, you cannot use extensions in it. You need to either use `logback-spring.xml` or define a `logging.config` property.

WARNING The extensions cannot be used with Logback's configuration scanning. If you attempt to do so, making changes to the configuration file results in an error similar to one of the following being logged:

```
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProperty], current ElementPath is [[configuration][springProperty]]
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProfile], current ElementPath is [[configuration][springProfile]]
```

Profile-specific Configuration

The `<springProfile>` tag lets you optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the `<configuration>` element. Use the `name` attribute to specify which profile accepts the configuration. The `<springProfile>` tag can contain a profile name (for example `staging`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example `production & (eu-central | eu-west)`. Check the Spring Framework reference guide for more details. The following listing shows three sample profiles:

```
<springProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
</springProfile>

<springProfile name="dev | staging">
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active
-->
</springProfile>

<springProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is not active -->
</springProfile>
```

Environment Properties

The `<springProperty>` tag lets you expose properties from the Spring `Environment` for use within Logback. Doing so can be useful if you want to access values from your `application.properties` file in your Logback configuration. The tag works in a similar way to Logback's standard `<property>` tag. However, rather than specifying a direct `value`, you specify the `source` of the property (from the `Environment`). If you need to store the property somewhere other than in `local` scope, you can use the `scope` attribute. If you need a fallback value (in case the property is not set in the `Environment`), you can use the `defaultValue` attribute. The following example shows how to expose properties for use within Logback:

```
<springProperty scope="context" name="fluentHost" source="myapp.fluentd.host"
        defaultValue="localhost"/>
<appender name="FLUENT" class="ch.qos.logback.more.appenders.DataFluentAppender">
    <remoteHost>${fluentHost}</remoteHost>
    ...
</appender>
```

NOTE	The `source` must be specified in kebab case (such as `my.property-name`). However, properties can be added to the `Environment` by using the relaxed rules.

7.4.10. Log4j2 Extensions

Spring Boot includes a number of extensions to Log4j2 that can help with advanced configuration.

You can use these extensions in any `log4j2-spring.xml` configuration file.

NOTE	Because the standard `log4j2.xml` configuration file is loaded too early, you cannot use extensions in it. You need to either use `log4j2-spring.xml` or define a `logging.config` property.

NOTE	The extensions supersede the Spring Boot support provided by Log4J. You should make sure not to include the `org.apache.logging.log4j:log4j-spring-boot` module in your build.

Profile-specific Configuration

The `<SpringProfile>` tag lets you optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the `<Configuration>` element. Use the `name` attribute to specify which profile accepts the configuration. The `<SpringProfile>` tag can contain a profile name (for example `staging`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example `production & (eu-central | eu-west)`. Check the Spring Framework reference guide for more details. The following listing shows three sample profiles:

```
<SpringProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
</SpringProfile>

<SpringProfile name="dev | staging">
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active
-->
</SpringProfile>

<SpringProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is not active -->
</SpringProfile>
```

Environment Properties Lookup

If you want to refer to properties from your Spring `Environment` within your Log4j2 configuration you can use `spring:` prefixed lookups. Doing so can be useful if you want to access values from your `application.properties` file in your Log4j2 configuration.

The following example shows how to set a Log4j2 property named `applicationName` that reads `spring.application.name` from the Spring `Environment`:

```
<Properties>
    <Property name="applicationName">${spring:spring.application.name}</Property>
</Properties>
```

NOTE	The lookup key should be specified in kebab case (such as `my.property-name`).

Log4j2 System Properties

Log4j2 supports a number of System Properties that can be used to configure various items. For example, the `log4j2.skipJansi` system property can be used to configure if the `ConsoleAppender` will try to use a Jansi output stream on Windows.

All system properties that are loaded after the Log4j2 initialization can be obtained from the Spring `Environment`. For example, you could add `log4j2.skipJansi=false` to your `application.properties` file to have the `ConsoleAppender` use Jansi on Windows.

NOTE	The Spring `Environment` is only considered when system properties and OS environment variables do not contain the value being loaded.

WARNING	System properties that are loaded during early Log4j2 initialization cannot reference the Spring `Environment`. For example, the property Log4j2 uses to allow the default Log4j2 implementation to be chosen is used before the Spring Environment is available.

7.5. Internationalization

Spring Boot supports localized messages so that your application can cater to users of different language preferences. By default, Spring Boot looks for the presence of a `messages` resource bundle at the root of the classpath.

NOTE	The auto-configuration applies when the default properties file for the configured resource bundle is available (`messages.properties` by default). If your resource bundle contains only language-specific properties files, you are required to add the default. If no properties file is found that matches any of the configured base names, there will be no auto-configured `MessageSource`.

The basename of the resource bundle as well as several other attributes can be configured using the `spring.messages` namespace, as shown in the following example:

Properties

```
spring.messages.basename=messages,config.i18n.messages
spring.messages.fallback-to-system-locale=false
```

Yaml

```
spring:
  messages:
    basename: "messages,config.i18n.messages"
    fallback-to-system-locale: false
```

TIP	`spring.messages.basename` supports comma-separated list of locations, either a package qualifier or a resource resolved from the classpath root.

See `MessageSourceProperties` for more supported options.

7.6. Aspect-Oriented Programming

Spring Boot provides auto-configuration for aspect-oriented programming (AOP). You can learn more about AOP with Spring in the Spring Framework reference documentation.

By default, Spring Boot's auto-configuration configures Spring AOP to use CGLib proxies. To use JDK proxies instead, set `configprop:spring.aop.proxy-target-class` to `false`.

If AspectJ is on the classpath, Spring Boot's auto-configuration will automatically enable AspectJ auto proxy such that `@EnableAspectJAutoProxy` is not required.

7.7. JSON

Spring Boot provides integration with three JSON mapping libraries:

- Gson
- Jackson
- JSON-B

Jackson is the preferred and default library.

7.7.1. Jackson

Auto-configuration for Jackson is provided and Jackson is part of `spring-boot-starter-json`. When Jackson is on the classpath an `ObjectMapper` bean is automatically configured. Several configuration properties are provided for customizing the configuration of the `ObjectMapper`.

Custom Serializers and Deserializers

If you use Jackson to serialize and deserialize JSON data, you might want to write your own `JsonSerializer` and `JsonDeserializer` classes. Custom serializers are usually registered with Jackson through a module, but Spring Boot provides an alternative `@JsonComponent` annotation that makes it easier to directly register Spring Beans.

You can use the `@JsonComponent` annotation directly on `JsonSerializer`, `JsonDeserializer` or `KeyDeserializer` implementations. You can also use it on classes that contain serializers/deserializers as inner classes, as shown in the following example:

Java

```
import java.io.IOException;

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.ObjectCodec;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonDeserializer;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.JsonSerializer;
import com.fasterxml.jackson.databind.SerializerProvider;

import org.springframework.boot.jackson.JsonComponent;

@JsonComponent
public class MyJsonComponent {

    public static class Serializer extends JsonSerializer<MyObject> {

        @Override
        public void serialize(MyObject value, JsonGenerator jgen, SerializerProvider
serializers) throws IOException {
            jgen.writeStartObject();
            jgen.writeStringField("name", value.getName());
            jgen.writeNumberField("age", value.getAge());
            jgen.writeEndObject();
        }

    }

    public static class Deserializer extends JsonDeserializer<MyObject> {

        @Override
        public MyObject deserialize(JsonParser jsonParser, DeserializationContext
ctxt) throws IOException {
            ObjectCodec codec = jsonParser.getCodec();
            JsonNode tree = codec.readTree(jsonParser);
            String name = tree.get("name").textValue();
            int age = tree.get("age").intValue();
            return new MyObject(name, age);
        }

    }

}
```

Kotlin

```
import com.fasterxml.jackson.core.JsonGenerator
import com.fasterxml.jackson.core.JsonParser
import com.fasterxml.jackson.core.JsonProcessingException
import com.fasterxml.jackson.databind.DeserializationContext
import com.fasterxml.jackson.databind.JsonDeserializer
import com.fasterxml.jackson.databind.JsonNode
import com.fasterxml.jackson.databind.JsonSerializer
import com.fasterxml.jackson.databind.SerializerProvider
import org.springframework.boot.jackson.JsonComponent
import java.io.IOException

@JsonComponent
class MyJsonComponent {

    class Serializer : JsonSerializer<MyObject>() {
        @Throws(IOException::class)
        override fun serialize(value: MyObject, jgen: JsonGenerator, serializers:
SerializerProvider) {
            jgen.writeStartObject()
            jgen.writeStringField("name", value.name)
            jgen.writeNumberField("age", value.age)
            jgen.writeEndObject()
        }
    }

    class Deserializer : JsonDeserializer<MyObject>() {
        @Throws(IOException::class, JsonProcessingException::class)
        override fun deserialize(jsonParser: JsonParser, ctxt:
DeserializationContext): MyObject {
            val codec = jsonParser.codec
            val tree = codec.readTree<JsonNode>(jsonParser)
            val name = tree["name"].textValue()
            val age = tree["age"].intValue()
            return MyObject(name, age)
        }
    }

}
```

All `@JsonComponent` beans in the `ApplicationContext` are automatically registered with Jackson. Because `@JsonComponent` is meta-annotated with `@Component`, the usual component-scanning rules apply.

Spring Boot also provides `JsonObjectSerializer` and `JsonObjectDeserializer` base classes that provide useful alternatives to the standard Jackson versions when serializing objects. See `JsonObjectSerializer` and `JsonObjectDeserializer` in the Javadoc for details.

The example above can be rewritten to use `JsonObjectSerializer`/`JsonObjectDeserializer` as follows:

Java

```
import java.io.IOException;

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.ObjectCodec;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.SerializerProvider;

import org.springframework.boot.jackson.JsonComponent;
import org.springframework.boot.jackson.JsonObjectDeserializer;
import org.springframework.boot.jackson.JsonObjectSerializer;

@JsonComponent
public class MyJsonComponent {

    public static class Serializer extends JsonObjectSerializer<MyObject> {

        @Override
        protected void serializeObject(MyObject value, JsonGenerator jgen,
SerializerProvider provider)
                throws IOException {
            jgen.writeStringField("name", value.getName());
            jgen.writeNumberField("age", value.getAge());
        }

    }

    public static class Deserializer extends JsonObjectDeserializer<MyObject> {

        @Override
        protected MyObject deserializeObject(JsonParser jsonParser,
DeserializationContext context, ObjectCodec codec,
                JsonNode tree) throws IOException {
            String name = nullSafeValue(tree.get("name"), String.class);
            int age = nullSafeValue(tree.get("age"), Integer.class);
            return new MyObject(name, age);
        }

    }

}
```

Kotlin

```
`object`

import com.fasterxml.jackson.core.JsonGenerator
import com.fasterxml.jackson.core.JsonParser
import com.fasterxml.jackson.core.ObjectCodec
import com.fasterxml.jackson.databind.DeserializationContext
import com.fasterxml.jackson.databind.JsonNode
import com.fasterxml.jackson.databind.SerializerProvider
import org.springframework.boot.jackson.JsonComponent
import org.springframework.boot.jackson.JsonObjectDeserializer
import org.springframework.boot.jackson.JsonObjectSerializer
import java.io.IOException

@JsonComponent
class MyJsonComponent {

    class Serializer : JsonObjectSerializer<MyObject>() {
        @Throws(IOException::class)
        override fun serializeObject(value: MyObject, jgen: JsonGenerator, provider:
SerializerProvider) {
            jgen.writeStringField("name", value.name)
            jgen.writeNumberField("age", value.age)
        }
    }

    class Deserializer : JsonObjectDeserializer<MyObject>() {
        @Throws(IOException::class)
        override fun deserializeObject(jsonParser: JsonParser, context:
DeserializationContext,
                codec: ObjectCodec, tree: JsonNode): MyObject {
            val name = nullSafeValue(tree["name"], String::class.java)
            val age = nullSafeValue(tree["age"], Int::class.java)
            return MyObject(name, age)
        }
    }

}
```

Mixins

Jackson has support for mixins that can be used to mix additional annotations into those already declared on a target class. Spring Boot's Jackson auto-configuration will scan your application's packages for classes annotated with `@JsonMixin` and register them with the auto-configured `ObjectMapper`. The registration is performed by Spring Boot's `JsonMixinModule`.

7.7.2. Gson

Auto-configuration for Gson is provided. When Gson is on the classpath a `Gson` bean is automatically

configured. Several `spring.gson.*` configuration properties are provided for customizing the configuration. To take more control, one or more `GsonBuilderCustomizer` beans can be used.

7.7.3. JSON-B

Auto-configuration for JSON-B is provided. When the JSON-B API and an implementation are on the classpath a `Jsonb` bean will be automatically configured. The preferred JSON-B implementation is Eclipse Yasson for which dependency management is provided.

7.8. Task Execution and Scheduling

In the absence of an `Executor` bean in the context, Spring Boot auto-configures a `ThreadPoolTaskExecutor` with sensible defaults that can be automatically associated to asynchronous task execution (`@EnableAsync`) and Spring MVC asynchronous request processing.

TIP

If you have defined a custom `Executor` in the context, regular task execution (that is `@EnableAsync`) will use it transparently but the Spring MVC support will not be configured as it requires an `AsyncTaskExecutor` implementation (named `applicationTaskExecutor`). Depending on your target arrangement, you could change your `Executor` into a `ThreadPoolTaskExecutor` or define both a `ThreadPoolTaskExecutor` and an `AsyncConfigurer` wrapping your custom `Executor`.

The auto-configured `TaskExecutorBuilder` allows you to easily create instances that reproduce what the auto-configuration does by default.

The thread pool uses 8 core threads that can grow and shrink according to the load. Those default settings can be fine-tuned using the `spring.task.execution` namespace, as shown in the following example:

Properties

```
spring.task.execution.pool.max-size=16
spring.task.execution.pool.queue-capacity=100
spring.task.execution.pool.keep-alive=10s
```

Yaml

```
spring:
  task:
    execution:
      pool:
        max-size: 16
        queue-capacity: 100
        keep-alive: "10s"
```

This changes the thread pool to use a bounded queue so that when the queue is full (100 tasks), the thread pool increases to maximum 16 threads. Shrinking of the pool is more aggressive as threads are reclaimed when they are idle for 10 seconds (rather than 60 seconds by default).

A `ThreadPoolTaskScheduler` can also be auto-configured if need to be associated to scheduled task execution (using `@EnableScheduling` for instance). The thread pool uses one thread by default and its settings can be fine-tuned using the `spring.task.scheduling` namespace, as shown in the following example:

Properties

```
spring.task.scheduling.thread-name-prefix=scheduling-
spring.task.scheduling.pool.size=2
```

Yaml

```
spring:
  task:
    scheduling:
      thread-name-prefix: "scheduling-"
      pool:
        size: 2
```

Both a `TaskExecutorBuilder` bean and a `TaskSchedulerBuilder` bean are made available in the context if a custom executor or scheduler needs to be created.

7.9. Testing

Spring Boot provides a number of utilities and annotations to help when testing your application. Test support is provided by two modules: `spring-boot-test` contains core items, and `spring-boot-test-autoconfigure` supports auto-configuration for tests.

Most developers use the `spring-boot-starter-test` "Starter", which imports both Spring Boot test modules as well as JUnit Jupiter, AssertJ, Hamcrest, and a number of other useful libraries.

TIP

If you have tests that use JUnit 4, JUnit 5's vintage engine can be used to run them. To use the vintage engine, add a dependency on `junit-vintage-engine`, as shown in the following example:

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

`hamcrest-core` is excluded in favor of `org.hamcrest:hamcrest` that is part of `spring-boot-starter-test`.

7.9.1. Test Scope Dependencies

The `spring-boot-starter-test` "Starter" (in the `test` scope) contains the following provided libraries:

- JUnit 5: The de-facto standard for unit testing Java applications.
- Spring Test & Spring Boot Test: Utilities and integration test support for Spring Boot applications.
- AssertJ: A fluent assertion library.
- Hamcrest: A library of matcher objects (also known as constraints or predicates).
- Mockito: A Java mocking framework.
- JSONassert: An assertion library for JSON.
- JsonPath: XPath for JSON.

We generally find these common libraries to be useful when writing tests. If these libraries do not suit your needs, you can add additional test dependencies of your own.

7.9.2. Testing Spring Applications

One of the major advantages of dependency injection is that it should make your code easier to unit test. You can instantiate objects by using the `new` operator without even involving Spring. You can also use *mock objects* instead of real dependencies.

Often, you need to move beyond unit testing and start integration testing (with a Spring `ApplicationContext`). It is useful to be able to perform integration testing without requiring deployment of your application or needing to connect to other infrastructure.

The Spring Framework includes a dedicated test module for such integration testing. You can declare a dependency directly to `org.springframework:spring-test` or use the `spring-boot-starter-test` "Starter" to pull it in transitively.

If you have not used the `spring-test` module before, you should start by reading the relevant section of the Spring Framework reference documentation.

7.9.3. Testing Spring Boot Applications

A Spring Boot application is a Spring `ApplicationContext`, so nothing very special has to be done to test it beyond what you would normally do with a vanilla Spring context.

NOTE	External properties, logging, and other features of Spring Boot are installed in the context by default only if you use `SpringApplication` to create it.

Spring Boot provides a `@SpringBootTest` annotation, which can be used as an alternative to the standard `spring-test` `@ContextConfiguration` annotation when you need Spring Boot features. The annotation works by creating the `ApplicationContext` used in your tests through `SpringApplication`. In addition to `@SpringBootTest` a number of other annotations are also provided for testing more

specific slices of an application.

TIP	If you are using JUnit 4, do not forget to also add `@RunWith(SpringRunner.class)` to your test, otherwise the annotations will be ignored. If you are using JUnit 5, there is no need to add the equivalent `@ExtendWith(SpringExtension.class)` as `@SpringBootTest` and the other `@…Test` annotations are already annotated with it.

By default, `@SpringBootTest` will not start a server. You can use the `webEnvironment` attribute of `@SpringBootTest` to further refine how your tests run:

- `MOCK`(Default) : Loads a web `ApplicationContext` and provides a mock web environment. Embedded servers are not started when using this annotation. If a web environment is not available on your classpath, this mode transparently falls back to creating a regular non-web `ApplicationContext`. It can be used in conjunction with `@AutoConfigureMockMvc` or `@AutoConfigureWebTestClient` for mock-based testing of your web application.
- `RANDOM_PORT`: Loads a `WebServerApplicationContext` and provides a real web environment. Embedded servers are started and listen on a random port.
- `DEFINED_PORT`: Loads a `WebServerApplicationContext` and provides a real web environment. Embedded servers are started and listen on a defined port (from your `application.properties`) or on the default port of `8080`.
- `NONE`: Loads an `ApplicationContext` by using `SpringApplication` but does not provide *any* web environment (mock or otherwise).

NOTE	If your test is `@Transactional`, it rolls back the transaction at the end of each test method by default. However, as using this arrangement with either `RANDOM_PORT` or `DEFINED_PORT` implicitly provides a real servlet environment, the HTTP client and server run in separate threads and, thus, in separate transactions. Any transaction initiated on the server does not roll back in this case.
NOTE	`@SpringBootTest` with `webEnvironment = WebEnvironment.RANDOM_PORT` will also start the management server on a separate random port if your application uses a different port for the management server.

Detecting Web Application Type

If Spring MVC is available, a regular MVC-based application context is configured. If you have only Spring WebFlux, we will detect that and configure a WebFlux-based application context instead.

If both are present, Spring MVC takes precedence. If you want to test a reactive web application in this scenario, you must set the `spring.main.web-application-type` property:

Java

```
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(properties = "spring.main.web-application-type=reactive")
class MyWebFluxTests {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.context.SpringBootTest

@SpringBootTest(properties = ["spring.main.web-application-type=reactive"])
class MyWebFluxTests {

    // ...

}
```

Detecting Test Configuration

If you are familiar with the Spring Test Framework, you may be used to using `@ContextConfiguration(classes=…)` in order to specify which Spring `@Configuration` to load. Alternatively, you might have often used nested `@Configuration` classes within your test.

When testing Spring Boot applications, this is often not required. Spring Boot's `@*Test` annotations search for your primary configuration automatically whenever you do not explicitly define one.

The search algorithm works up from the package that contains the test until it finds a class annotated with `@SpringBootApplication` or `@SpringBootConfiguration`. As long as you structured your code in a sensible way, your main configuration is usually found.

If you use a test annotation to test a more specific slice of your application, you should avoid adding configuration settings that are specific to a particular area on the main method's application class.

NOTE

The underlying component scan configuration of `@SpringBootApplication` defines exclude filters that are used to make sure slicing works as expected. If you are using an explicit `@ComponentScan` directive on your `@SpringBootApplication`-annotated class, be aware that those filters will be disabled. If you are using slicing, you should define them again.

If you want to customize the primary configuration, you can use a nested `@TestConfiguration` class. Unlike a nested `@Configuration` class, which would be used instead of your application's primary configuration, a nested `@TestConfiguration` class is used in addition to your application's primary configuration.

NOTE	Spring's test framework caches application contexts between tests. Therefore, as long as your tests share the same configuration (no matter how it is discovered), the potentially time-consuming process of loading the context happens only once.

Using the Test Configuration Main Method

Typically the test configuration discovered by `@SpringBootTest` will be your main `@SpringBootApplication`. In most well structured applications, this configuration class will also include the `main` method used to launch the application.

For example, the following is a very common code pattern for a typical Spring Boot application:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import
org.springframework.boot.docs.using.structuringyourcode.locatingthemainclass.MyApplica
tion
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

In the example above, the `main` method doesn't do anything other than delegate to `SpringApplication.run`. It is, however, possible to have a more complex `main` method that applies customizations before calling `SpringApplication.run`.

For example, here is an application that changes the banner mode and sets additional profiles:

Java

```
import org.springframework.boot.Banner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(MyApplication.class);
        application.setBannerMode(Banner.Mode.OFF);
        application.setAdditionalProfiles("myprofile");
        application.run(args);
    }

}
```

Kotlin

```
import org.springframework.boot.Banner
import org.springframework.boot.runApplication
import org.springframework.boot.autoconfigure.SpringBootApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args) {
        setBannerMode(Banner.Mode.OFF)
        setAdditionalProfiles("myprofile");
    }
}
```

Since customizations in the `main` method can affect the resulting `ApplicationContext`, it's possible that you might also want to use the `main` method to create the `ApplicationContext` used in your tests. By default, `@SpringBootTest` will not call your `main` method, and instead the class itself is used directly to create the `ApplicationContext`

If you want to change this behavior, you can change the `useMainMethod` attribute of `@SpringBootTest` to `UseMainMethod.ALWAYS` or `UseMainMethod.WHEN_AVAILABLE`. When set to `ALWAYS`, the test will fail if no `main` method can be found. When set to `WHEN_AVAILABLE` the `main` method will be used if it is available, otherwise the standard loading mechanism will be used.

For example, the following test will invoke the `main` method of `MyApplication` in order to create the `ApplicationContext`. If the main method sets additional profiles then those will be active when the `ApplicationContext` starts.

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.UseMainMethod;

@SpringBootTest(useMainMethod = UseMainMethod.ALWAYS)
class MyApplicationTests {

    @Test
    void exampleTest() {
        // ...
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.UseMainMethod
import org.springframework.context.annotation.Import

@SpringBootTest(useMainMethod = UseMainMethod.ALWAYS)
class MyApplicationTests {

    @Test
    fun exampleTest() {
        // ...
    }

}
```

Excluding Test Configuration

If your application uses component scanning (for example, if you use `@SpringBootApplication` or `@ComponentScan`), you may find top-level configuration classes that you created only for specific tests accidentally get picked up everywhere.

As we have seen earlier, `@TestConfiguration` can be used on an inner class of a test to customize the primary configuration. `@TestConfiguration` can also be used on a top-level class. Doing so indicates that the class should not be picked up by scanning. You can then import the class explicitly where it is required, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Import;

@SpringBootTest
@Import(MyTestsConfiguration.class)
class MyTests {

    @Test
    void exampleTest() {
        // ...
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.context.annotation.Import

@SpringBootTest
@Import(MyTestsConfiguration::class)
class MyTests {

    @Test
    fun exampleTest() {
        // ...
    }

}
```

NOTE	If you directly use `@ComponentScan` (that is, not through `@SpringBootApplication`) you need to register the `TypeExcludeFilter` with it. See the Javadoc for details.
NOTE	An imported `@TestConfiguration` is processed earlier than an inner-class `@TestConfiguration` and an imported `@TestConfiguration` will be processed before any configuration found through component scanning. Generally speaking, this difference in ordering has no noticeable effect but it is something to be aware of if you're relying on bean overriding.

Using Application Arguments

If your application expects arguments, you can have `@SpringBootTest` inject them using the `args` attribute.

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.test.context.SpringBootTest;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(args = "--app.test=one")
class MyApplicationArgumentTests {

    @Test
    void applicationArgumentsPopulated(@Autowired ApplicationArguments args) {
        assertThat(args.getOptionNames()).containsOnly("app.test");
        assertThat(args.getOptionValues("app.test")).containsOnly("one");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.ApplicationArguments
import org.springframework.boot.test.context.SpringBootTest

@SpringBootTest(args = ["--app.test=one"])
class MyApplicationArgumentTests {

    @Test
    fun applicationArgumentsPopulated(@Autowired args: ApplicationArguments) {
        assertThat(args.optionNames).containsOnly("app.test")
        assertThat(args.getOptionValues("app.test")).containsOnly("one")
    }

}
```

Testing With a Mock Environment

By default, `@SpringBootTest` does not start the server but instead sets up a mock environment for testing web endpoints.

With Spring MVC, we can query our web endpoints using `MockMvc` or `WebTestClient`, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.reactive.server.WebTestClient;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
class MyMockMvcTests {

    @Test
    void testWithMockMvc(@Autowired MockMvc mvc) throws Exception {

mvc.perform(get("/")).andExpect(status().isOk()).andExpect(content().string("Hello
World"));
    }

    // If Spring WebFlux is on the classpath, you can drive MVC tests with a
WebTestClient
    @Test
    void testWithWebTestClient(@Autowired WebTestClient webClient) {
        webClient
                .get().uri("/")
                .exchange()
                .expectStatus().isOk()
                .expectBody(String.class).isEqualTo("Hello World");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
import org.springframework.test.web.servlet.result.MockMvcResultMatchers

@SpringBootTest
@AutoConfigureMockMvc
class MyMockMvcTests {

    @Test
    fun testWithMockMvc(@Autowired mvc: MockMvc) {

mvc.perform(MockMvcRequestBuilders.get("/")).andExpect(MockMvcResultMatchers.status().
isOk)
            .andExpect(MockMvcResultMatchers.content().string("Hello World"))
    }

    // If Spring WebFlux is on the classpath, you can drive MVC tests with a
WebTestClient

    @Test
    fun testWithWebTestClient(@Autowired webClient: WebTestClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Hello World")
    }

}
```

TIP If you want to focus only on the web layer and not start a complete `ApplicationContext`, consider using `@WebMvcTest` instead.

With Spring WebFlux endpoints, you can use `WebTestClient` as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestClient;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.reactive.server.WebTestClient;

@SpringBootTest
@AutoConfigureWebTestClient
class MyMockWebTestClientTests {

    @Test
    void exampleTest(@Autowired WebTestClient webClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello World");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestClient
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody

@SpringBootTest
@AutoConfigureWebTestClient
class MyMockWebTestClientTests {

    @Test
    fun exampleTest(@Autowired webClient: WebTestClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Hello World")
    }

}
```

Testing within a mocked environment is usually faster than running with a full servlet container. However, since mocking occurs at the Spring MVC layer, code that relies on lower-level servlet container behavior cannot be directly tested with MockMvc.

TIP

For example, Spring Boot's error handling is based on the "error page" support provided by the servlet container. This means that, whilst you can test your MVC layer throws and handles exceptions as expected, you cannot directly test that a specific custom error page is rendered. If you need to test these lower-level concerns, you can start a fully running server as described in the next section.

Testing With a Running Server

If you need to start a full running server, we recommend that you use random ports. If you use `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`, an available port is picked at random each time your test runs.

The `@LocalServerPort` annotation can be used to inject the actual port used into your test. For convenience, tests that need to make REST calls to the started server can additionally `@Autowire` a `WebTestClient`, which resolves relative links to the running server and comes with a dedicated API for verifying responses, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.test.web.reactive.server.WebTestClient;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortWebTestClientTests {

    @Test
    void exampleTest(@Autowired WebTestClient webClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello World");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortWebTestClientTests {

    @Test
    fun exampleTest(@Autowired webClient: WebTestClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Hello World")
    }

}
```

TIP `WebTestClient` can also used with a mock environment, removing the need for a running server, by annotating your test class with `@AutoConfigureWebTestClient`.

This setup requires `spring-webflux` on the classpath. If you can not or will not add webflux, Spring Boot also provides a `TestRestTemplate` facility:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortTestRestTemplateTests {

    @Test
    void exampleTest(@Autowired TestRestTemplate restTemplate) {
        String body = restTemplate.getForObject("/", String.class);
        assertThat(body).isEqualTo("Hello World");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.boot.test.web.client.TestRestTemplate

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortTestRestTemplateTests {

    @Test
    fun exampleTest(@Autowired restTemplate: TestRestTemplate) {
        val body = restTemplate.getForObject("/", String::class.java)
        assertThat(body).isEqualTo("Hello World")
    }

}
```

Customizing WebTestClient

To customize the `WebTestClient` bean, configure a `WebTestClientBuilderCustomizer` bean. Any such beans are called with the `WebTestClient.Builder` that is used to create the `WebTestClient`.

Using JMX

As the test context framework caches context, JMX is disabled by default to prevent identical components to register on the same domain. If such test needs access to an `MBeanServer`, consider marking it dirty as well:

Java

```
import javax.management.MBeanServer;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.DirtiesContext;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(properties = "spring.jmx.enabled=true")
@DirtiesContext
class MyJmxTests {

    @Autowired
    private MBeanServer mBeanServer;

    @Test
    void exampleTest() {
        assertThat(this.mBeanServer.getDomains()).contains("java.lang");
        // ...
    }

}
```

Kotlin

```
import javax.management.MBeanServer

import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.annotation.DirtiesContext

@SpringBootTest(properties = ["spring.jmx.enabled=true"])
@DirtiesContext
class MyJmxTests(@Autowired val mBeanServer: MBeanServer) {

    @Test
    fun exampleTest() {
        assertThat(mBeanServer.domains).contains("java.lang")
        // ...
    }

}
```

Using Metrics

Regardless of your classpath, meter registries, except the in-memory backed, are not auto-configured when using `@SpringBootTest`.

If you need to export metrics to a different backend as part of an integration test, annotate it with `@AutoConfigureObservability`.

Using Tracing

Regardless of your classpath, tracing is not auto-configured when using `@SpringBootTest`.

If you need tracing as part of an integration test, annotate it with `@AutoConfigureObservability`.

Mocking and Spying Beans

When running tests, it is sometimes necessary to mock certain components within your application context. For example, you may have a facade over some remote service that is unavailable during development. Mocking can also be useful when you want to simulate failures that might be hard to trigger in a real environment.

Spring Boot includes a `@MockBean` annotation that can be used to define a Mockito mock for a bean inside your `ApplicationContext`. You can use the annotation to add new beans or replace a single existing bean definition. The annotation can be used directly on test classes, on fields within your test, or on `@Configuration` classes and fields. When used on a field, the instance of the created mock is also injected. Mock beans are automatically reset after each test method.

NOTE

If your test uses one of Spring Boot's test annotations (such as `@SpringBootTest`), this feature is automatically enabled. To use this feature with a different arrangement, listeners must be explicitly added, as shown in the following example:

Java

```
import
org.springframework.boot.test.mock.mockito.MockitoTestExecutionListener;
import
org.springframework.boot.test.mock.mockito.ResetMocksTestExecutionListen
er;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.TestExecutionListeners;

@ContextConfiguration(classes = MyConfig.class)
@TestExecutionListeners({ MockitoTestExecutionListener.class,
ResetMocksTestExecutionListener.class })
class MyTests {

    // ...

}
```

Kotlin

```
import
org.springframework.boot.test.mock.mockito.MockitoTestExecutionListener
import
org.springframework.boot.test.mock.mockito.ResetMocksTestExecutionListen
er
import org.springframework.test.context.ContextConfiguration
import org.springframework.test.context.TestExecutionListeners

@ContextConfiguration(classes = [MyConfig::class])
@TestExecutionListeners(
    MockitoTestExecutionListener::class,
    ResetMocksTestExecutionListener::class
)
class MyTests {

    // ...

}
```

The following example replaces an existing `RemoteService` bean with a mock implementation:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.given;

@SpringBootTest
class MyTests {

    @Autowired
    private Reverser reverser;

    @MockBean
    private RemoteService remoteService;

    @Test
    void exampleTest() {
        given(this.remoteService.getValue()).willReturn("spring");
        String reverse = this.reverser.getReverseValue(); // Calls injected
RemoteService
        assertThat(reverse).isEqualTo("gnirps");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.mock.mockito.MockBean

@SpringBootTest
class MyTests(@Autowired val reverser: Reverser, @MockBean val remoteService:
RemoteService) {

    @Test
    fun exampleTest() {
        given(remoteService.value).willReturn("spring")
        val reverse = reverser.reverseValue // Calls injected RemoteService
        assertThat(reverse).isEqualTo("gnirps")
    }

}
```

NOTE `@MockBean` cannot be used to mock the behavior of a bean that is exercised during application context refresh. By the time the test is executed, the application context refresh has completed and it is too late to configure the mocked behavior. We recommend using a `@Bean` method to create and configure the mock in this situation.

Additionally, you can use `@SpyBean` to wrap any existing bean with a Mockito `spy`. See the Javadoc for full details.

NOTE While Spring's test framework caches application contexts between tests and reuses a context for tests sharing the same configuration, the use of `@MockBean` or `@SpyBean` influences the cache key, which will most likely increase the number of contexts.

TIP If you are using `@SpyBean` to spy on a bean with `@Cacheable` methods that refer to parameters by name, your application must be compiled with `-parameters`. This ensures that the parameter names are available to the caching infrastructure once the bean has been spied upon.

TIP When you are using `@SpyBean` to spy on a bean that is proxied by Spring, you may need to remove Spring's proxy in some situations, for example when setting expectations using `given` or `when`. Use `AopTestUtils.getTargetObject(yourProxiedSpy)` to do so.

Auto-configured Tests

Spring Boot's auto-configuration system works well for applications but can sometimes be a little too much for tests. It often helps to load only the parts of the configuration that are required to test a "slice" of your application. For example, you might want to test that Spring MVC controllers are

mapping URLs correctly, and you do not want to involve database calls in those tests, or you might want to test JPA entities, and you are not interested in the web layer when those tests run.

The `spring-boot-test-autoconfigure` module includes a number of annotations that can be used to automatically configure such "slices". Each of them works in a similar way, providing a `@…Test` annotation that loads the `ApplicationContext` and one or more `@AutoConfigure…` annotations that can be used to customize auto-configuration settings.

NOTE	Each slice restricts component scan to appropriate components and loads a very restricted set of auto-configuration classes. If you need to exclude one of them, most `@…Test` annotations provide an `excludeAutoConfiguration` attribute. Alternatively, you can use `@ImportAutoConfiguration#exclude`.
NOTE	Including multiple "slices" by using several `@…Test` annotations in one test is not supported. If you need multiple "slices", pick one of the `@…Test` annotations and include the `@AutoConfigure…` annotations of the other "slices" by hand.
TIP	It is also possible to use the `@AutoConfigure…` annotations with the standard `@SpringBootTest` annotation. You can use this combination if you are not interested in "slicing" your application but you want some of the auto-configured test beans.

Auto-configured JSON Tests

To test that object JSON serialization and deserialization is working as expected, you can use the `@JsonTest` annotation. `@JsonTest` auto-configures the available supported JSON mapper, which can be one of the following libraries:

- Jackson `ObjectMapper`, any `@JsonComponent` beans and any Jackson `Module`s
- `Gson`
- `Jsonb`

TIP	A list of the auto-configurations that are enabled by `@JsonTest` can be found in the appendix.

If you need to configure elements of the auto-configuration, you can use the `@AutoConfigureJsonTesters` annotation.

Spring Boot includes AssertJ-based helpers that work with the JSONAssert and JsonPath libraries to check that JSON appears as expected. The `JacksonTester`, `GsonTester`, `JsonbTester`, and `BasicJsonTester` classes can be used for Jackson, Gson, Jsonb, and Strings respectively. Any helper fields on the test class can be `@Autowired` when using `@JsonTest`. The following example shows a test class for Jackson:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.json.JsonTest;
import org.springframework.boot.test.json.JacksonTester;

import static org.assertj.core.api.Assertions.assertThat;

@JsonTest
class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    void serialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda", "Civic");
        // Assert against a `.json` file in the same package as the test
        assertThat(this.json.write(details)).isEqualToJson("expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");

assertThat(this.json.write(details)).extractingJsonPathStringValue("@.make").isEqualTo
("Honda");
    }

    @Test
    void deserialize() throws Exception {
        String content = "{\"make\":\"Ford\",\"model\":\"Focus\"}";
        assertThat(this.json.parse(content)).isEqualTo(new VehicleDetails("Ford",
"Focus"));
        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.json.JsonTest
import org.springframework.boot.test.json.JacksonTester

@JsonTest
class MyJsonTests(@Autowired val json: JacksonTester<VehicleDetails>) {

    @Test
    fun serialize() {
        val details = VehicleDetails("Honda", "Civic")
        // Assert against a `.json` file in the same package as the test
        assertThat(json.write(details)).isEqualToJson("expected.json")
        // Or use JSON path based assertions
        assertThat(json.write(details)).hasJsonPathStringValue("@.make")

assertThat(json.write(details)).extractingJsonPathStringValue("@.make").isEqualTo("Hon
da")
    }

    @Test
    fun deserialize() {
        val content = "{\"make\":\"Ford\",\"model\":\"Focus\"}"
        assertThat(json.parse(content)).isEqualTo(VehicleDetails("Ford", "Focus"))
        assertThat(json.parseObject(content).make).isEqualTo("Ford")
    }

}
```

NOTE JSON helper classes can also be used directly in standard unit tests. To do so, call the `initFields` method of the helper in your `@Before` method if you do not use `@JsonTest`.

If you use Spring Boot's AssertJ-based helpers to assert on a number value at a given JSON path, you might not be able to use `isEqualTo` depending on the type. Instead, you can use AssertJ's `satisfies` to assert that the value matches the given condition. For instance, the following example asserts that the actual number is a float value close to `0.15` within an offset of `0.01`.

Java

```
@Test
void someTest() throws Exception {
    SomeObject value = new SomeObject(0.152f);

assertThat(this.json.write(value)).extractingJsonPathNumberValue("@.test.numberValue")
        .satisfies((number) -> assertThat(number.floatValue()).isCloseTo(0.15f,
within(0.01f)));
}
```

Kotlin

```
@Test
fun someTest() {
    val value = SomeObject(0.152f)
    assertThat(json.write(value)).extractingJsonPathNumberValue("@.test.numberValue")
        .satisfies(ThrowingConsumer { number ->
            assertThat(number.toFloat()).isCloseTo(0.15f, within(0.01f))
        })
}
```

Auto-configured Spring MVC Tests

To test whether Spring MVC controllers are working as expected, use the `@WebMvcTest` annotation. `@WebMvcTest` auto-configures the Spring MVC infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `Filter`, `HandlerInterceptor`, `WebMvcConfigurer`, `WebMvcRegistrations`, and `HandlerMethodArgumentResolver`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@WebMvcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP	A list of the auto-configuration settings that are enabled by `@WebMvcTest` can be found in the appendix.

TIP	If you need to register extra components, such as the Jackson `Module`, you can import additional configuration classes by using `@Import` on your test.

Often, `@WebMvcTest` is limited to a single controller and is used in combination with `@MockBean` to provide mock implementations for required collaborators.

`@WebMvcTest` also auto-configures `MockMvc`. Mock MVC offers a powerful way to quickly test MVC controllers without needing to start a full HTTP server.

TIP	You can also auto-configure `MockMvc` in a non-`@WebMvcTest` (such as `@SpringBootTest`) by annotating it with `@AutoConfigureMockMvc`. The following example uses `MockMvc`:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import static org.mockito.BDDMockito.given;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(UserVehicleController.class)
class MyControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));
        this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk())
            .andExpect(content().string("Honda Civic"));
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.boot.test.mock.mockito.MockBean
import org.springframework.http.MediaType
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
import org.springframework.test.web.servlet.result.MockMvcResultMatchers

@WebMvcTest(UserVehicleController::class)
class MyControllerTests(@Autowired val mvc: MockMvc) {

    @MockBean
    lateinit var userVehicleService: UserVehicleService

    @Test
    fun testExample() {
        given(userVehicleService.getVehicleDetails("sboot"))
            .willReturn(VehicleDetails("Honda", "Civic"))

mvc.perform(MockMvcRequestBuilders.get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
            .andExpect(MockMvcResultMatchers.status().isOk)
            .andExpect(MockMvcResultMatchers.content().string("Honda Civic"))
    }

}
```

TIP If you need to configure elements of the auto-configuration (for example, when servlet filters should be applied) you can use attributes in the `@AutoConfigureMockMvc` annotation.

If you use HtmlUnit and Selenium, auto-configuration also provides an HtmlUnit `WebClient` bean and/or a Selenium `WebDriver` bean. The following example uses HtmlUnit:

Java

```
import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.HtmlPage;
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.given;

@WebMvcTest(UserVehicleController.class)
class MyHtmlUnitTests {

    @Autowired
    private WebClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot")).willReturn(new
VehicleDetails("Honda", "Civic"));
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
        assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
    }

}
```

Kotlin

```
import com.gargoylesoftware.htmlunit.WebClient
import com.gargoylesoftware.htmlunit.html.HtmlPage
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.boot.test.mock.mockito.MockBean

@WebMvcTest(UserVehicleController::class)
class MyHtmlUnitTests(@Autowired val webClient: WebClient) {

    @MockBean
    lateinit var userVehicleService: UserVehicleService

    @Test
    fun testExample() {

given(userVehicleService.getVehicleDetails("sboot")).willReturn(VehicleDetails("Honda"
, "Civic"))
        val page = webClient.getPage<HtmlPage>("/sboot/vehicle.html")
        assertThat(page.body.textContent).isEqualTo("Honda Civic")
    }

}
```

NOTE	By default, Spring Boot puts `WebDriver` beans in a special "scope" to ensure that the driver exits after each test and that a new instance is injected. If you do not want this behavior, you can add `@Scope("singleton")` to your `WebDriver` `@Bean` definition.

WARNING	The `webDriver` scope created by Spring Boot will replace any user defined scope of the same name. If you define your own `webDriver` scope you may find it stops working when you use `@WebMvcTest`.

If you have Spring Security on the classpath, `@WebMvcTest` will also scan `WebSecurityConfigurer` beans. Instead of disabling security completely for such tests, you can use Spring Security's test support. More details on how to use Spring Security's `MockMvc` support can be found in this *Testing With Spring Security* how-to section.

TIP	Sometimes writing Spring MVC tests is not enough; Spring Boot can help you run full end-to-end tests with an actual server.

Auto-configured Spring WebFlux Tests

To test that Spring WebFlux controllers are working as expected, you can use the `@WebFluxTest` annotation. `@WebFluxTest` auto-configures the Spring WebFlux infrastructure and limits scanned

beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `WebFilter`, and `WebFluxConfigurer`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@WebFluxTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP A list of the auto-configurations that are enabled by `@WebFluxTest` can be found in the appendix.

TIP If you need to register extra components, such as Jackson `Module`, you can import additional configuration classes using `@Import` on your test.

Often, `@WebFluxTest` is limited to a single controller and used in combination with the `@MockBean` annotation to provide mock implementations for required collaborators.

`@WebFluxTest` also auto-configures `WebTestClient`, which offers a powerful way to quickly test WebFlux controllers without needing to start a full HTTP server.

TIP You can also auto-configure `WebTestClient` in a non-`@WebFluxTest` (such as `@SpringBootTest`) by annotating it with `@AutoConfigureWebTestClient`. The following example shows a class that uses both `@WebFluxTest` and a `WebTestClient`:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.reactive.server.WebTestClient;

import static org.mockito.BDDMockito.given;

@WebFluxTest(UserVehicleController.class)
class MyControllerTests {

    @Autowired
    private WebTestClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));

this.webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN).exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Honda Civic");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest
import org.springframework.boot.test.mock.mockito.MockBean
import org.springframework.http.MediaType
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody

@WebFluxTest(UserVehicleController::class)
class MyControllerTests(@Autowired val webClient: WebTestClient) {

    @MockBean
    lateinit var userVehicleService: UserVehicleService

    @Test
    fun testExample() {
        given(userVehicleService.getVehicleDetails("sboot"))
            .willReturn(VehicleDetails("Honda", "Civic"))
        webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN).exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Honda Civic")
    }

}
```

TIP This setup is only supported by WebFlux applications as using `WebTestClient` in a mocked web application only works with WebFlux at the moment.

NOTE `@WebFluxTest` cannot detect routes registered through the functional web framework. For testing `RouterFunction` beans in the context, consider importing your `RouterFunction` yourself by using `@Import` or by using `@SpringBootTest`.

NOTE `@WebFluxTest` cannot detect custom security configuration registered as a `@Bean` of type `SecurityWebFilterChain`. To include that in your test, you will need to import the configuration that registers the bean by using `@Import` or by using `@SpringBootTest`.

TIP Sometimes writing Spring WebFlux tests is not enough; Spring Boot can help you run full end-to-end tests with an actual server.

Auto-configured Spring GraphQL Tests

Spring GraphQL offers a dedicated testing support module; you'll need to add it to your project:

Maven

```
<dependencies>
  <dependency>
    <groupId>org.springframework.graphql</groupId>
    <artifactId>spring-graphql-test</artifactId>
    <scope>test</scope>
  </dependency>
  <!-- Unless already present in the compile scope -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Gradle

```
dependencies {
  testImplementation("org.springframework.graphql:spring-graphql-test")
  // Unless already present in the implementation configuration
  testImplementation("org.springframework.boot:spring-boot-starter-webflux")
}
```

This testing module ships the GraphQlTester. The tester is heavily used in test, so be sure to become familiar with using it. There are `GraphQlTester` variants and Spring Boot will auto-configure them depending on the type of tests:

- the `ExecutionGraphQlServiceTester` performs tests on the server side, without a client nor a transport
- the `HttpGraphQlTester` performs tests with a client that connects to a server, with or without a live server

Spring Boot helps you to test your Spring GraphQL Controllers with the `@GraphQlTest` annotation. `@GraphQlTest` auto-configures the Spring GraphQL infrastructure, without any transport nor server being involved. This limits scanned beans to `@Controller`, `RuntimeWiringConfigurer`, `JsonComponent`, `Converter`, `GenericConverter`, `DataFetcherExceptionResolver`, `Instrumentation` and `GraphQlSourceBuilderCustomizer`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@GraphQlTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP A list of the auto-configurations that are enabled by `@GraphQlTest` can be found in the appendix.

Often, `@GraphQlTest` is limited to a set of controllers and used in combination with the `@MockBean` annotation to provide mock implementations for required collaborators.

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.docs.web.graphql.runtimewiring.GreetingController;
import org.springframework.boot.test.autoconfigure.graphql.GraphQlTest;
import org.springframework.graphql.test.tester.GraphQlTester;

@GraphQlTest(GreetingController.class)
class GreetingControllerTests {

    @Autowired
    private GraphQlTester graphQlTester;

    @Test
    void shouldGreetWithSpecificName() {
        this.graphQlTester.document("{ greeting(name: \"Alice\") } ")
            .execute()
            .path("greeting")
            .entity(String.class)
            .isEqualTo("Hello, Alice!");
    }

    @Test
    void shouldGreetWithDefaultName() {
        this.graphQlTester.document("{ greeting } ")
            .execute()
            .path("greeting")
            .entity(String.class)
            .isEqualTo("Hello, Spring!");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.docs.web.graphql.runtimewiring.GreetingController
import org.springframework.boot.test.autoconfigure.graphql.GraphQlTest
import org.springframework.graphql.test.tester.GraphQlTester

@GraphQlTest(GreetingController::class)
internal class GreetingControllerTests {

    @Autowired
    lateinit var graphQlTester: GraphQlTester

    @Test
    fun shouldGreetWithSpecificName() {
        graphQlTester.document("{ greeting(name: \"Alice\") }
").execute().path("greeting").entity(String::class.java)
                .isEqualTo("Hello, Alice!")
    }

    @Test
    fun shouldGreetWithDefaultName() {
        graphQlTester.document("{ greeting }
").execute().path("greeting").entity(String::class.java)
                .isEqualTo("Hello, Spring!")
    }

}
```

`@SpringBootTest` tests are full integration tests and involve the entire application. When using a random or defined port, a live server is configured and an `HttpGraphQlTester` bean is contributed automatically so you can use it to test your server. When a MOCK environment is configured, you can also request an `HttpGraphQlTester` bean by annotating your test class with `@AutoConfigureHttpGraphQlTester`:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.graphql.tester.AutoConfigureHttpGraphQlTes
ter;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.graphql.test.tester.HttpGraphQlTester;

@AutoConfigureHttpGraphQlTester
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
class GraphQlIntegrationTests {

    @Test
    void shouldGreetWithSpecificName(@Autowired HttpGraphQlTester graphQlTester) {
        HttpGraphQlTester authenticatedTester = graphQlTester.mutate()
            .webTestClient((client) -> client.defaultHeaders((headers) ->
headers.setBasicAuth("admin", "ilovespring")))
            .build();
        authenticatedTester.document("{ greeting(name: \"Alice\") } ")
            .execute()
            .path("greeting")
            .entity(String.class)
            .isEqualTo("Hello, Alice!");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.graphql.tester.AutoConfigureHttpGraphQlTes
ter
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.graphql.test.tester.HttpGraphQlTester
import org.springframework.http.HttpHeaders
import org.springframework.test.web.reactive.server.WebTestClient

@AutoConfigureHttpGraphQlTester
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
class GraphQlIntegrationTests {

    @Test
    fun shouldGreetWithSpecificName(@Autowired graphQlTester: HttpGraphQlTester) {
        val authenticatedTester = graphQlTester.mutate()
            .webTestClient { client: WebTestClient.Builder ->
                client.defaultHeaders { headers: HttpHeaders ->
                    headers.setBasicAuth("admin", "ilovespring")
                }
            }.build()
        authenticatedTester.document("{ greeting(name: \"Alice\") } ").execute()
            .path("greeting").entity(String::class.java).isEqualTo("Hello, Alice!")
    }
}
```

Auto-configured Data Cassandra Tests

You can use `@DataCassandraTest` to test Cassandra applications. By default, it configures a `CassandraTemplate`, scans for `@Table` classes, and configures Spring Data Cassandra repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataCassandraTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Cassandra with Spring Boot, see "Cassandra".)

> **TIP** A list of the auto-configuration settings that are enabled by `@DataCassandraTest` can be found in the appendix.

The following example shows a typical setup for using Cassandra tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.cassandra.DataCassandraTest;

@DataCassandraTest
class MyDataCassandraTests {

    @Autowired
    private SomeRepository repository;

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.cassandra.DataCassandraTest

@DataCassandraTest
class MyDataCassandraTests(@Autowired val repository: SomeRepository)
```

Auto-configured Data Couchbase Tests

You can use `@DataCouchbaseTest` to test Couchbase applications. By default, it configures a `CouchbaseTemplate` or `ReactiveCouchbaseTemplate`, scans for `@Document` classes, and configures Spring Data Couchbase repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataCouchbaseTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Couchbase with Spring Boot, see "Couchbase", earlier in this chapter.)

TIP	A list of the auto-configuration settings that are enabled by `@DataCouchbaseTest` can be found in the appendix.

The following example shows a typical setup for using Couchbase tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.couchbase.DataCouchbaseTest;

@DataCouchbaseTest
class MyDataCouchbaseTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.couchbase.DataCouchbaseTest

@DataCouchbaseTest
class MyDataCouchbaseTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

Auto-configured Data Elasticsearch Tests

You can use `@DataElasticsearchTest` to test Elasticsearch applications. By default, it configures an `ElasticsearchRestTemplate`, scans for `@Document` classes, and configures Spring Data Elasticsearch repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataElasticsearchTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Elasticsearch with Spring Boot, see "Elasticsearch", earlier in this chapter.)

TIP	A list of the auto-configuration settings that are enabled by `@DataElasticsearchTest` can be found in the appendix.

The following example shows a typical setup for using Elasticsearch tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.data.elasticsearch.DataElasticsearchTest;

@DataElasticsearchTest
class MyDataElasticsearchTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.data.elasticsearch.DataElasticsearchTest

@DataElasticsearchTest
class MyDataElasticsearchTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

Auto-configured Data JPA Tests

You can use the `@DataJpaTest` annotation to test JPA applications. By default, it scans for `@Entity` classes and configures Spring Data JPA repositories. If an embedded database is available on the classpath, it configures one as well. SQL queries are logged by default by setting the `spring.jpa.show-sql` property to `true`. This can be disabled using the `showSql` attribute of the annotation.

Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataJpaTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP A list of the auto-configuration settings that are enabled by `@DataJpaTest` can be found in the appendix.

By default, data JPA tests are transactional and roll back at the end of each test. See the relevant section in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class as follows:

Java

```
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyNonTransactionalTests {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest
import org.springframework.transaction.annotation.Propagation
import org.springframework.transaction.annotation.Transactional

@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyNonTransactionalTests {

    // ...

}
```

Data JPA tests may also inject a `TestEntityManager` bean, which provides an alternative to the standard JPA `EntityManager` that is specifically designed for tests.

TIP

`TestEntityManager` can also be auto-configured to any of your Spring-based test class by adding `@AutoConfigureTestEntityManager`. When doing so, make sure that your test is running in a transaction, for instance by adding `@Transactional` on your test class or method.

A `JdbcTemplate` is also available if you need that. The following example shows the `@DataJpaTest` annotation in use:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;

import static org.assertj.core.api.Assertions.assertThat;

@DataJpaTest
class MyRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    void testExample() {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getEmployeeNumber()).isEqualTo("1234");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager

@DataJpaTest
class MyRepositoryTests(@Autowired val entityManager: TestEntityManager, @Autowired
val repository: UserRepository) {

    @Test
    fun testExample() {
        entityManager.persist(User("sboot", "1234"))
        val user = repository.findByUsername("sboot")
        assertThat(user?.username).isEqualTo("sboot")
        assertThat(user?.employeeNumber).isEqualTo("1234")
    }

}
```

In-memory embedded databases generally work well for tests, since they are fast and do not require any installation. If, however, you prefer to run tests against a real database you can use the `@AutoConfigureTestDatabase` annotation, as shown in the following example:

Java

```
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import
org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase.Replace;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
class MyRepositoryTests {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest

@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class MyRepositoryTests {

    // ...

}
```

Auto-configured JDBC Tests

`@JdbcTest` is similar to `@DataJpaTest` but is for tests that only require a `DataSource` and do not use Spring Data JDBC. By default, it configures an in-memory embedded database and a `JdbcTemplate`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@JdbcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP	A list of the auto-configurations that are enabled by `@JdbcTest` can be found in the appendix.

By default, JDBC tests are transactional and roll back at the end of each test. See the relevant section in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class, as follows:

Java

```
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyTransactionalTests {

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest
import org.springframework.transaction.annotation.Propagation
import org.springframework.transaction.annotation.Transactional

@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyTransactionalTests
```

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `@DataJpaTest`. (See "Auto-configured Data JPA Tests".)

Auto-configured Data JDBC Tests

`@DataJdbcTest` is similar to `@JdbcTest` but is for tests that use Spring Data JDBC repositories. By default, it configures an in-memory embedded database, a `JdbcTemplate`, and Spring Data JDBC repositories. Only `AbstractJdbcConfiguration` subclasses are scanned when the `@DataJdbcTest` annotation is used, regular `@Component` and `@ConfigurationProperties` beans are not scanned. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP	A list of the auto-configurations that are enabled by `@DataJdbcTest` can be found in the appendix.

By default, Data JDBC tests are transactional and roll back at the end of each test. See the relevant section in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole test class as shown in the JDBC example.

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `@DataJpaTest`. (See "Auto-configured Data JPA Tests".)

Auto-configured Data R2DBC Tests

`@DataR2dbcTest` is similar to `@DataJdbcTest` but is for tests that use Spring Data R2DBC repositories. By default, it configures an in-memory embedded database, an `R2dbcEntityTemplate`, and Spring Data R2DBC repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned

when the `@DataR2dbcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP	A list of the auto-configurations that are enabled by `@DataR2dbcTest` can be found in the appendix.

By default, Data R2DBC tests are not transactional.

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `@DataJpaTest`. (See "Auto-configured Data JPA Tests".)

Auto-configured jOOQ Tests

You can use `@JooqTest` in a similar fashion as `@JdbcTest` but for jOOQ-related tests. As jOOQ relies heavily on a Java-based schema that corresponds with the database schema, the existing `DataSource` is used. If you want to replace it with an in-memory database, you can use `@AutoConfigureTestDatabase` to override those settings. (For more about using jOOQ with Spring Boot, see "Using jOOQ".) Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@JooqTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP	A list of the auto-configurations that are enabled by `@JooqTest` can be found in the appendix.

`@JooqTest` configures a `DSLContext`. The following example shows the `@JooqTest` annotation in use:

Java

```
import org.jooq.DSLContext;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;

@JooqTest
class MyJooqTests {

    @Autowired
    private DSLContext dslContext;

    // ...

}
```

Kotlin

```
import org.jooq.DSLContext
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.jooq.JooqTest

@JooqTest
class MyJooqTests(@Autowired val dslContext: DSLContext) {

    // ...

}
```

JOOQ tests are transactional and roll back at the end of each test by default. If that is not what you want, you can disable transaction management for a test or for the whole test class as shown in the JDBC example.

Auto-configured Data MongoDB Tests

You can use `@DataMongoTest` to test MongoDB applications. By default, it configures a `MongoTemplate`, scans for `@Document` classes, and configures Spring Data MongoDB repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataMongoTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using MongoDB with Spring Boot, see "MongoDB".)

TIP	A list of the auto-configuration settings that are enabled by `@DataMongoTest` can be found in the appendix.

The following class shows the `@DataMongoTest` annotation in use:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.data.mongodb.core.MongoTemplate;

@DataMongoTest
class MyDataMongoDbTests {

    @Autowired
    private MongoTemplate mongoTemplate;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest
import org.springframework.data.mongodb.core.MongoTemplate

@DataMongoTest
class MyDataMongoDbTests(@Autowired val mongoTemplate: MongoTemplate) {

    // ...

}
```

Auto-configured Data Neo4j Tests

You can use `@DataNeo4jTest` to test Neo4j applications. By default, it scans for `@Node` classes, and configures Spring Data Neo4j repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataNeo4jTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Neo4J with Spring Boot, see "Neo4j".)

> **TIP** A list of the auto-configuration settings that are enabled by `@DataNeo4jTest` can be found in the appendix.

The following example shows a typical setup for using Neo4J tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;

@DataNeo4jTest
class MyDataNeo4jTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest

@DataNeo4jTest
class MyDataNeo4jTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

By default, Data Neo4j tests are transactional and roll back at the end of each test. See the relevant section in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class, as follows:

Java

```
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyDataNeo4jTests {

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest
import org.springframework.transaction.annotation.Propagation
import org.springframework.transaction.annotation.Transactional

@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyDataNeo4jTests
```

NOTE	Transactional tests are not supported with reactive access. If you are using this style, you must configure `@DataNeo4jTest` tests as described above.

Auto-configured Data Redis Tests

You can use `@DataRedisTest` to test Redis applications. By default, it scans for `@RedisHash` classes and configures Spring Data Redis repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataRedisTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Redis with Spring Boot, see "Redis".)

TIP A list of the auto-configuration settings that are enabled by `@DataRedisTest` can be found in the appendix.

The following example shows the `@DataRedisTest` annotation in use:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest;

@DataRedisTest
class MyDataRedisTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest

@DataRedisTest
class MyDataRedisTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

Auto-configured Data LDAP Tests

You can use `@DataLdapTest` to test LDAP applications. By default, it configures an in-memory embedded LDAP (if available), configures an `LdapTemplate`, scans for `@Entry` classes, and configures Spring Data LDAP repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataLdapTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using LDAP with Spring Boot, see "LDAP".)

TIP A list of the auto-configuration settings that are enabled by `@DataLdapTest` can be found in the appendix.

The following example shows the `@DataLdapTest` annotation in use:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.ldap.core.LdapTemplate;

@DataLdapTest
class MyDataLdapTests {

    @Autowired
    private LdapTemplate ldapTemplate;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest
import org.springframework.ldap.core.LdapTemplate

@DataLdapTest
class MyDataLdapTests(@Autowired val ldapTemplate: LdapTemplate) {

    // ...

}
```

In-memory embedded LDAP generally works well for tests, since it is fast and does not require any developer installation. If, however, you prefer to run tests against a real LDAP server, you should exclude the embedded LDAP auto-configuration, as shown in the following example:

Java

```
import
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;

@DataLdapTest(excludeAutoConfiguration = EmbeddedLdapAutoConfiguration.class)
class MyDataLdapTests {

    // ...

}
```

Kotlin

```
import
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest

@DataLdapTest(excludeAutoConfiguration = [EmbeddedLdapAutoConfiguration::class])
class MyDataLdapTests {

    // ...

}
```

Auto-configured REST Clients

You can use the `@RestClientTest` annotation to test REST clients. By default, it auto-configures Jackson, GSON, and Jsonb support, configures a `RestTemplateBuilder`, and adds support for `MockRestServiceServer`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@RestClientTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP	A list of the auto-configuration settings that are enabled by `@RestClientTest` can be found in the appendix.

The specific beans that you want to test should be specified by using the `value` or `components` attribute of `@RestClientTest`, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.client.RestClientTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.client.MockRestServiceServer;

import static org.assertj.core.api.Assertions.assertThat;
import static
org.springframework.test.web.client.match.MockRestRequestMatchers.requestTo;
import static
org.springframework.test.web.client.response.MockRestResponseCreators.withSuccess;

@RestClientTest(RemoteVehicleDetailsService.class)
class MyRestClientTests {

    @Autowired
    private RemoteVehicleDetailsService service;

    @Autowired
    private MockRestServiceServer server;

    @Test
    void getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() {

this.server.expect(requestTo("/greet/details")).andRespond(withSuccess("hello",
MediaType.TEXT_PLAIN));
        String greeting = this.service.callRestService();
        assertThat(greeting).isEqualTo("hello");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.client.RestClientTest
import org.springframework.http.MediaType
import org.springframework.test.web.client.MockRestServiceServer
import org.springframework.test.web.client.match.MockRestRequestMatchers
import org.springframework.test.web.client.response.MockRestResponseCreators

@RestClientTest(RemoteVehicleDetailsService::class)
class MyRestClientTests(
    @Autowired val service: RemoteVehicleDetailsService,
    @Autowired val server: MockRestServiceServer) {

    @Test
    fun getVehicleDetailsWhenResultIsSuccessShouldReturnDetails(): Unit {
        server.expect(MockRestRequestMatchers.requestTo("/greet/details"))
            .andRespond(MockRestResponseCreators.withSuccess("hello",
MediaType.TEXT_PLAIN))
        val greeting = service.callRestService()
        assertThat(greeting).isEqualTo("hello")
    }

}
```

Auto-configured Spring REST Docs Tests

You can use the `@AutoConfigureRestDocs` annotation to use Spring REST Docs in your tests with Mock MVC, REST Assured, or WebTestClient. It removes the need for the JUnit extension in Spring REST Docs.

`@AutoConfigureRestDocs` can be used to override the default output directory (`target/generated-snippets` if you are using Maven or `build/generated-snippets` if you are using Gradle). It can also be used to configure the host, scheme, and port that appears in any documented URIs.

Auto-configured Spring REST Docs Tests With Mock MVC

`@AutoConfigureRestDocs` customizes the `MockMvc` bean to use Spring REST Docs when testing servlet-based web applications. You can inject it by using `@Autowired` and use it in your tests as you normally would when using Mock MVC and Spring REST Docs, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(UserController.class)
@AutoConfigureRestDocs
class MyUserDocumentationTests {

    @Autowired
    private MockMvc mvc;

    @Test
    void listUsers() throws Exception {
        this.mvc.perform(get("/users").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk())
            .andDo(document("list-users"));
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.http.MediaType
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentation
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
import org.springframework.test.web.servlet.result.MockMvcResultMatchers

@WebMvcTest(UserController::class)
@AutoConfigureRestDocs
class MyUserDocumentationTests(@Autowired val mvc: MockMvc) {

    @Test
    fun listUsers() {
        mvc.perform(MockMvcRequestBuilders.get("/users").accept(MediaType.TEXT_PLAIN))
            .andExpect(MockMvcResultMatchers.status().isOk)
            .andDo(MockMvcRestDocumentation.document("list-users"))
    }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, you can use a `RestDocsMockMvcConfigurationCustomizer` bean, as shown in the following example:

Java

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsMockMvcConfigurationCusto
mizer;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentationConfigurer;
import org.springframework.restdocs.templates.TemplateFormats;

@TestConfiguration(proxyBeanMethods = false)
public class MyRestDocsConfiguration implements RestDocsMockMvcConfigurationCustomizer
{

    @Override
    public void customize(MockMvcRestDocumentationConfigurer configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}
```

Kotlin

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsMockMvcConfigurationCusto
mizer
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentationConfigurer
import org.springframework.restdocs.templates.TemplateFormats

@TestConfiguration(proxyBeanMethods = false)
class MyRestDocsConfiguration : RestDocsMockMvcConfigurationCustomizer {

    override fun customize(configurer: MockMvcRestDocumentationConfigurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown())
    }

}
```

If you want to make use of Spring REST Docs support for a parameterized output directory, you can create a `RestDocumentationResultHandler` bean. The auto-configuration calls `alwaysDo` with this result handler, thereby causing each `MockMvc` call to automatically generate the default snippets. The following example shows a `RestDocumentationResultHandler` being defined:

Java

```
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentation;
import org.springframework.restdocs.mockmvc.RestDocumentationResultHandler;

@TestConfiguration(proxyBeanMethods = false)
public class MyResultHandlerConfiguration {

    @Bean
    public RestDocumentationResultHandler restDocumentation() {
        return MockMvcRestDocumentation.document("{method-name}");
    }

}
```

Kotlin

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.context.annotation.Bean
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentation
import org.springframework.restdocs.mockmvc.RestDocumentationResultHandler

@TestConfiguration(proxyBeanMethods = false)
class MyResultHandlerConfiguration {

    @Bean
    fun restDocumentation(): RestDocumentationResultHandler {
        return MockMvcRestDocumentation.document("{method-name}")
    }

}
```

Auto-configured Spring REST Docs Tests With WebTestClient

`@AutoConfigureRestDocs` can also be used with `WebTestClient` when testing reactive web applications. You can inject it by using `@Autowired` and use it in your tests as you normally would when using `@WebFluxTest` and Spring REST Docs, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.test.web.reactive.server.WebTestClient;

import static
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation.document;

@WebFluxTest
@AutoConfigureRestDocs
class MyUsersDocumentationTests {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    void listUsers() {
        this.webTestClient
            .get().uri("/")
        .exchange()
        .expectStatus()
            .isOk()
        .expectBody()
            .consumeWith(document("list-users"));
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest
import org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation
import org.springframework.test.web.reactive.server.WebTestClient

@WebFluxTest
@AutoConfigureRestDocs
class MyUsersDocumentationTests(@Autowired val webTestClient: WebTestClient) {

    @Test
    fun listUsers() {
        webTestClient
            .get().uri("/")
            .exchange()
            .expectStatus()
            .isOk
            .expectBody()
            .consumeWith(WebTestClientRestDocumentation.document("list-users"))
    }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, you can use a `RestDocsWebTestClientConfigurationCustomizer` bean, as shown in the following example:

Java

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsWebTestClientConfiguratio
nCustomizer;
import org.springframework.boot.test.context.TestConfiguration;
import
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentationConfigurer;

@TestConfiguration(proxyBeanMethods = false)
public class MyRestDocsConfiguration implements
RestDocsWebTestClientConfigurationCustomizer {

    @Override
    public void customize(WebTestClientRestDocumentationConfigurer configurer) {
        configurer.snippets().withEncoding("UTF-8");
    }

}
```

Kotlin

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsWebTestClientConfiguratio
nCustomizer
import org.springframework.boot.test.context.TestConfiguration
import
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentationConfigurer

@TestConfiguration(proxyBeanMethods = false)
class MyRestDocsConfiguration : RestDocsWebTestClientConfigurationCustomizer {

    override fun customize(configurer: WebTestClientRestDocumentationConfigurer) {
        configurer.snippets().withEncoding("UTF-8")
    }

}
```

If you want to make use of Spring REST Docs support for a parameterized output directory, you can use a `WebTestClientBuilderCustomizer` to configure a consumer for every entity exchange result. The following example shows such a `WebTestClientBuilderCustomizer` being defined:

Java

```
import org.springframework.boot.test.context.TestConfiguration;
import
org.springframework.boot.test.web.reactive.server.WebTestClientBuilderCustomizer;
import org.springframework.context.annotation.Bean;

import static
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation.document;

@TestConfiguration(proxyBeanMethods = false)
public class MyWebTestClientBuilderCustomizerConfiguration {

    @Bean
    public WebTestClientBuilderCustomizer restDocumentation() {
        return (builder) -> builder.entityExchangeResultConsumer(document("{method-
name}"));
    }

}
```

Kotlin

```
import org.springframework.boot.test.context.TestConfiguration
import
org.springframework.boot.test.web.reactive.server.WebTestClientBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation
import org.springframework.test.web.reactive.server.WebTestClient

@TestConfiguration(proxyBeanMethods = false)
class MyWebTestClientBuilderCustomizerConfiguration {

    @Bean
    fun restDocumentation(): WebTestClientBuilderCustomizer {
        return WebTestClientBuilderCustomizer { builder: WebTestClient.Builder ->
            builder.entityExchangeResultConsumer(
                WebTestClientRestDocumentation.document("{method-name}")
            )
        }
    }

}
```

Auto-configured Spring REST Docs Tests With REST Assured

`@AutoConfigureRestDocs` makes a `RequestSpecification` bean, preconfigured to use Spring REST Docs, available to your tests. You can inject it by using `@Autowired` and use it in your tests as you normally would when using REST Assured and Spring REST Docs, as shown in the following example:

Java

```
import io.restassured.specification.RequestSpecification;
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.server.LocalServerPort;

import static io.restassured.RestAssured.given;
import static org.hamcrest.Matchers.is;
import static
org.springframework.restdocs.restassured.RestAssuredRestDocumentation.document;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
class MyUserDocumentationTests {

    @Test
    void listUsers(@Autowired RequestSpecification documentationSpec, @LocalServerPort
int port) {
        given(documentationSpec)
            .filter(document("list-users"))
        .when()
            .port(port)
            .get("/")
        .then().assertThat()
            .statusCode(is(200));
    }

}
```

Kotlin

```
import io.restassured.RestAssured
import io.restassured.specification.RequestSpecification
import org.hamcrest.Matchers
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.boot.test.web.server.LocalServerPort
import org.springframework.restdocs.restassured.RestAssuredRestDocumentation

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
class MyUserDocumentationTests {

    @Test
    fun listUsers(@Autowired documentationSpec: RequestSpecification?,
@LocalServerPort port: Int) {
        RestAssured.given(documentationSpec)
            .filter(RestAssuredRestDocumentation.document("list-users"))
            .`when`()
            .port(port)["/"]
            .then().assertThat()
            .statusCode(Matchers.`is`(200))
    }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, a `RestDocsRestAssuredConfigurationCustomizer` bean can be used, as shown in the following example:

Java

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsRestAssuredConfigurationC
ustomizer;
import org.springframework.boot.test.context.TestConfiguration;
import
org.springframework.restdocs.restassured.RestAssuredRestDocumentationConfigurer;
import org.springframework.restdocs.templates.TemplateFormats;

@TestConfiguration(proxyBeanMethods = false)
public class MyRestDocsConfiguration implements
RestDocsRestAssuredConfigurationCustomizer {

    @Override
    public void customize(RestAssuredRestDocumentationConfigurer configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}
```

Kotlin

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsRestAssuredConfigurationC
ustomizer
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.restdocs.restassured.RestAssuredRestDocumentationConfigurer
import org.springframework.restdocs.templates.TemplateFormats

@TestConfiguration(proxyBeanMethods = false)
class MyRestDocsConfiguration : RestDocsRestAssuredConfigurationCustomizer {

    override fun customize(configurer: RestAssuredRestDocumentationConfigurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown())
    }

}
```

Auto-configured Spring Web Services Tests

Auto-configured Spring Web Services Client Tests

You can use `@WebServiceClientTest` to test applications that call web services using the Spring Web Services project. By default, it configures a mock `WebServiceServer` bean and automatically customizes your `WebServiceTemplateBuilder`. (For more about using Web Services with Spring Boot, see "Web Services".)

TIP A list of the auto-configuration settings that are enabled by `@WebServiceClientTest` can be found in the appendix.

The following example shows the `@WebServiceClientTest` annotation in use:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.webservices.client.WebServiceClientTest;
import org.springframework.ws.test.client.MockWebServiceServer;
import org.springframework.xml.transform.StringSource;

import static org.assertj.core.api.Assertions.assertThat;
import static org.springframework.ws.test.client.RequestMatchers.payload;
import static org.springframework.ws.test.client.ResponseCreators.withPayload;

@WebServiceClientTest(SomeWebService.class)
class MyWebServiceClientTests {

    @Autowired
    private MockWebServiceServer server;

    @Autowired
    private SomeWebService someWebService;

    @Test
    void mockServerCall() {
        this.server
            .expect(payload(new StringSource("<request/>")))
            .andRespond(withPayload(new
StringSource("<response><status>200</status></response>")));
        assertThat(this.someWebService.test())
            .extracting(Response::getStatus)
            .isEqualTo(200);
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.webservices.client.WebServiceClientTest
import org.springframework.ws.test.client.MockWebServiceServer
import org.springframework.ws.test.client.RequestMatchers
import org.springframework.ws.test.client.ResponseCreators
import org.springframework.xml.transform.StringSource

@WebServiceClientTest(SomeWebService::class)
class MyWebServiceClientTests(@Autowired val server: MockWebServiceServer, @Autowired
val someWebService: SomeWebService) {

    @Test
    fun mockServerCall() {
        server
            .expect(RequestMatchers.payload(StringSource("<request/>")))

.andRespond(ResponseCreators.withPayload(StringSource("<response><status>200</status><
/response>")))

assertThat(this.someWebService.test()).extracting(Response::status).isEqualTo(200)
    }

}
```

Auto-configured Spring Web Services Server Tests

You can use `@WebServiceServerTest` to test applications that implement web services using the Spring Web Services project. By default, it configures a `MockWebServiceClient` bean that can be used to call your web service endpoints. (For more about using Web Services with Spring Boot, see "Web Services".)

> **TIP** A list of the auto-configuration settings that are enabled by `@WebServiceServerTest` can be found in the appendix.

The following example shows the `@WebServiceServerTest` annotation in use:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.webservices.server.WebServiceServerTest;
import org.springframework.ws.test.server.MockWebServiceClient;
import org.springframework.ws.test.server.RequestCreators;
import org.springframework.ws.test.server.ResponseMatchers;
import org.springframework.xml.transform.StringSource;

@WebServiceServerTest(ExampleEndpoint.class)
class MyWebServiceServerTests {

    @Autowired
    private MockWebServiceClient client;

    @Test
    void mockServerCall() {
        this.client
            .sendRequest(RequestCreators.withPayload(new
StringSource("<ExampleRequest/>")))
            .andExpect(ResponseMatchers.payload(new
StringSource("<ExampleResponse>42</ExampleResponse>")));
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.webservices.server.WebServiceServerTest
import org.springframework.ws.test.server.MockWebServiceClient
import org.springframework.ws.test.server.RequestCreators
import org.springframework.ws.test.server.ResponseMatchers
import org.springframework.xml.transform.StringSource

@WebServiceServerTest(ExampleEndpoint::class)
class MyWebServiceServerTests(@Autowired val client: MockWebServiceClient) {

    @Test
    fun mockServerCall() {
        client

.sendRequest(RequestCreators.withPayload(StringSource("<ExampleRequest/>")))

.andExpect(ResponseMatchers.payload(StringSource("<ExampleResponse>42</ExampleResponse
>")))
    }

}
```

Additional Auto-configuration and Slicing

Each slice provides one or more `@AutoConfigure…` annotations that namely defines the auto-configurations that should be included as part of a slice. Additional auto-configurations can be added on a test-by-test basis by creating a custom `@AutoConfigure…` annotation or by adding `@ImportAutoConfiguration` to the test as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.ImportAutoConfiguration;
import
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration;
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;

@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration.class)
class MyJdbcTests {

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.ImportAutoConfiguration
import org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest

@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration::class)
class MyJdbcTests
```

NOTE	Make sure to not use the regular `@Import` annotation to import auto-configurations as they are handled in a specific way by Spring Boot.

Alternatively, additional auto-configurations can be added for any use of a slice annotation by registering them in a file stored in `META-INF/spring` as shown in the following example:

META-INF/spring/org.springframework.boot.test.autoconfigure.jdbc.JdbcTest.imports

```
com.example.IntegrationAutoConfiguration
```

In this example, the `com.example.IntegrationAutoConfiguration` is enabled on every test annotated with `@JdbcTest`.

TIP	You can use comments with `#` in this file.
TIP	A slice or `@AutoConfigure⋯` annotation can be customized this way as long as it is meta-annotated with `@ImportAutoConfiguration`.

User Configuration and Slicing

If you structure your code in a sensible way, your `@SpringBootApplication` class is used by default as the configuration of your tests.

It then becomes important not to litter the application's main class with configuration settings that are specific to a particular area of its functionality.

Assume that you are using Spring Data MongoDB, you rely on the auto-configuration for it, and you have enabled auditing. You could define your `@SpringBootApplication` as follows:

Java

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.mongodb.config.EnableMongoAuditing;

@SpringBootApplication
@EnableMongoAuditing
public class MyApplication {

    // ...

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.data.mongodb.config.EnableMongoAuditing

@SpringBootApplication
@EnableMongoAuditing
class MyApplication {

    // ...

}
```

Because this class is the source configuration for the test, any slice test actually tries to enable Mongo auditing, which is definitely not what you want to do. A recommended approach is to move that area-specific configuration to a separate `@Configuration` class at the same level as your application, as shown in the following example:

Java

```
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.config.EnableMongoAuditing;

@Configuration(proxyBeanMethods = false)
@EnableMongoAuditing
public class MyMongoConfiguration {

    // ...

}
```

Kotlin

```
import org.springframework.context.annotation.Configuration
import org.springframework.data.mongodb.config.EnableMongoAuditing;

@Configuration(proxyBeanMethods = false)
@EnableMongoAuditing
class MyMongoConfiguration {

    // ...

}
```

NOTE Depending on the complexity of your application, you may either have a single `@Configuration` class for your customizations or one class per domain area. The latter approach lets you enable it in one of your tests, if necessary, with the `@Import` annotation. See this how-to section for more details on when you might want to enable specific `@Configuration` classes for slice tests.

Test slices exclude `@Configuration` classes from scanning. For example, for a `@WebMvcTest`, the following configuration will not include the given `WebMvcConfigurer` bean in the application context loaded by the test slice:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration(proxyBeanMethods = false)
public class MyWebConfiguration {

    @Bean
    public WebMvcConfigurer testConfigurer() {
        return new WebMvcConfigurer() {
            // ...
        };
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Configuration(proxyBeanMethods = false)
class MyWebConfiguration {

    @Bean
    fun testConfigurer(): WebMvcConfigurer {
        return object : WebMvcConfigurer {
            // ...
        }
    }

}
```

The configuration below will, however, cause the custom `WebMvcConfigurer` to be loaded by the test slice.

Java

```
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Component
public class MyWebMvcConfigurer implements WebMvcConfigurer {

    // ...

}
```

Kotlin

```
import org.springframework.stereotype.Component
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Component
class MyWebMvcConfigurer : WebMvcConfigurer {

    // ...

}
```

Another source of confusion is classpath scanning. Assume that, while you structured your code in a sensible way, you need to scan an additional package. Your application may resemble the following code:

Java

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@ComponentScan({ "com.example.app", "com.example.another" })
public class MyApplication {

    // ...

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.context.annotation.ComponentScan

@SpringBootApplication
@ComponentScan("com.example.app", "com.example.another")
class MyApplication {

    // ...

}
```

Doing so effectively overrides the default component scan directive with the side effect of scanning those two packages regardless of the slice that you chose. For instance, a `@DataJpaTest` seems to suddenly scan components and user configurations of your application. Again, moving the custom directive to a separate class is a good way to fix this issue.

TIP	If this is not an option for you, you can create a `@SpringBootConfiguration` somewhere in the hierarchy of your test so that it is used instead. Alternatively, you can specify a source for your test, which disables the behavior of finding a default one.

Using Spock to Test Spring Boot Applications

Spock 2.2 or later can be used to test a Spring Boot application. To do so, add a dependency on a `-groovy-4.0` version of Spock's `spock-spring` module to your application's build. `spock-spring` integrates Spring's test framework into Spock. See the documentation for Spock's Spring module for further details.

7.9.4. Testcontainers

The Testcontainers library provides a way to manage services running inside Docker containers. It integrates with JUnit, allowing you to write a test class that can start up a container before any of the tests run. Testcontainers is especially useful for writing integration tests that talk to a real backend service such as MySQL, MongoDB, Cassandra and others.

Testcontainers can be used in a Spring Boot test as follows:

Java

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import org.springframework.boot.test.context.SpringBootTest;

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>("neo4j:5");

    @Test
    void myTest() {
        // ...
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.testcontainers.containers.Neo4jContainer
import org.testcontainers.junit.jupiter.Container
import org.testcontainers.junit.jupiter.Testcontainers

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Test
    fun myTest() {
        // ...
    }

    companion object {
        @Container
        val neo4j = Neo4jContainer("neo4j:5")
    }

}
```

This will start up a docker container running Neo4j (if Docker is running locally) before any of the

tests are run. In most cases, you will need to configure the application to connect to the service running in the container.

Service Connections

A service connection is a connection to any remote service. Spring Boot's auto-configuration can consume the details of a service connection and use them to establish a connection to a remote service. When doing so, the connection details take precedence over any connection-related configuration properties.

When using Testcontainers, connection details can be automatically created for a service running in a container by annotating the container field in the test class.

Java

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Container
    @ServiceConnection
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>("neo4j:5");

    @Test
    void myTest() {
        // ...
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.testcontainers.service.connection.ServiceConnection
import org.testcontainers.containers.Neo4jContainer
import org.testcontainers.junit.jupiter.Container
import org.testcontainers.junit.jupiter.Testcontainers

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Test
    fun myTest() {
        // ...
    }

    companion object {

        @Container
        @ServiceConnection
        val neo4j = Neo4jContainer("neo4j:5")

    }

}
```

Thanks to `@ServiceConnection`, the above configuration allows Neo4j-related beans in the application to communicate with Neo4j running inside the Testcontainers-managed Docker container. This is done by automatically defining a `Neo4jConnectionDetails` bean which is then used by the Neo4j auto-configuration, overriding any connection-related configuration properties.

NOTE You'll need to add the `spring-boot-testcontainers` module as a test dependency in order to use service connections with Testcontainers.

Service connection annotations are processed by `ContainerConnectionDetailsFactory` classes registered with `spring.factories`. A `ContainerConnectionDetailsFactory` can create a `ConnectionDetails` bean based on a specific `Container` subclass, or the Docker image name.

The following service connection factories are provided in the `spring-boot-testcontainers` jar:

Connection Details	Matched on
`CassandraConnectionDetails`	Containers of type `CassandraContainer`
`CouchbaseConnectionDetails`	Containers of type `CouchbaseContainer`
`ElasticsearchConnectionDetails`	Containers of type `ElasticsearchContainer`
`FlywayConnectionDetails`	Containers of type `JdbcDatabaseContainer`

Connection Details	Matched on
`JdbcConnectionDetails`	Containers of type `JdbcDatabaseContainer`
`KafkaConnectionDetails`	Containers of type `KafkaContainer` or `RedpandaContainer`
`LiquibaseConnectionDetails`	Containers of type `JdbcDatabaseContainer`
`MongoConnectionDetails`	Containers of type `MongoDBContainer`
`Neo4jConnectionDetails`	Containers of type `Neo4jContainer`
`R2dbcConnectionDetails`	Containers of type `MariaDBContainer`, `MSSQLServerContainer`, `MySQLContainer`, `OracleContainer`, or `PostgreSQLContainer`
`RabbitConnectionDetails`	Containers of type `RabbitMQContainer`
`RedisConnectionDetails`	Containers named "redis"
`ZipkinConnectionDetails`	Containers named "openzipkin/zipkin"

TIP

By default all applicable connection details beans will be created for a given `Container`. For example, a `PostgreSQLContainer` will create both `JdbcConnectionDetails` and `R2dbcConnectionDetails`.

If you want to create only a subset of the applicable types, you can use the `type` attribute of `@ServiceConnection`.

By default `Container.getDockerImageName()` is used to obtain the name used to find connection details. This works as long as Spring Boot is able to get the instance of the `Container`, which is the case when using a `static` field like in the example above.

If you're using a `@Bean` method, Spring Boot won't call the bean method to get the Docker image name, because this would cause eager initialization issues. Instead, the return type of the bean method is used to find out which connection detail should be used. This works as long as you're using typed containers, e.g. `Neo4jContainer` or `RabbitMQContainer`. This stops working if you're using `GenericContainer`, e.g. with Redis, as shown in the following example:

Java

```
import org.testcontainers.containers.GenericContainer;

import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;
import org.springframework.context.annotation.Bean;

@TestConfiguration(proxyBeanMethods = false)
public class MyRedisConfiguration {

    @Bean
    @ServiceConnection(name = "redis")
    public GenericContainer<?> redisContainer() {
        return new GenericContainer<>("redis:7");
    }

}
```

Kotlin

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.service.connection.ServiceConnection
import org.springframework.context.annotation.Bean;
import org.testcontainers.containers.GenericContainer

@TestConfiguration(proxyBeanMethods = false)
class MyRedisConfiguration {

    @Bean
    @ServiceConnection(name = "redis")
    fun redisContainer(): GenericContainer<*> {
        return GenericContainer("redis:7")
    }

}
```

Spring Boot can't tell from `GenericContainer` which container image is used, so the `name` attribute from `@ServiceConnection` must be used to provide that hint.

You can also can use the `name` attribute of `@ServiceConnection` to override which connection detail will be used, for example when using custom images. If you are using the Docker image `registry.mycompany.com/mirror/myredis`, you'd use `@ServiceConnection(name="redis")` to ensure `RedisConnectionDetails` are created.

Dynamic Properties

A slightly more verbose but also more flexible alternative to service connections is `@DynamicPropertySource`. A static `@DynamicPropertySource` method allows adding dynamic property

values to the Spring Environment.

Java

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>("neo4j:5");

    @Test
    void myTest() {
        // ...
    }

    @DynamicPropertySource
    static void neo4jProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.neo4j.uri", neo4j::getBoltUrl);
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.context.DynamicPropertyRegistry
import org.springframework.test.context.DynamicPropertySource
import org.testcontainers.containers.Neo4jContainer
import org.testcontainers.junit.jupiter.Container
import org.testcontainers.junit.jupiter.Testcontainers

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Test
    fun myTest() {
        // ...
    }

    companion object {

        @Container
        val neo4j = Neo4jContainer("neo4j:5")

        @DynamicPropertySource
        fun neo4jProperties(registry: DynamicPropertyRegistry) {
            registry.add("spring.neo4j.uri") { neo4j.boltUrl }
        }

    }

}
```

The above configuration allows Neo4j-related beans in the application to communicate with Neo4j running inside the Testcontainers-managed Docker container.

7.9.5. Test Utilities

A few test utility classes that are generally useful when testing your application are packaged as part of `spring-boot`.

ConfigDataApplicationContextInitializer

`ConfigDataApplicationContextInitializer` is an `ApplicationContextInitializer` that you can apply to your tests to load Spring Boot `application.properties` files. You can use it when you do not need the full set of features provided by `@SpringBootTest`, as shown in the following example:

Java

```
import org.springframework.boot.test.context.ConfigDataApplicationContextInitializer;
import org.springframework.test.context.ContextConfiguration;

@ContextConfiguration(classes = Config.class, initializers =
ConfigDataApplicationContextInitializer.class)
class MyConfigFileTests {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.context.ConfigDataApplicationContextInitializer
import org.springframework.test.context.ContextConfiguration

@ContextConfiguration(classes = [Config::class], initializers =
[ConfigDataApplicationContextInitializer::class])
class MyConfigFileTests {

    // ...

}
```

NOTE	Using `ConfigDataApplicationContextInitializer` alone does not provide support for `@Value("${…}")` injection. Its only job is to ensure that `application.properties` files are loaded into Spring's `Environment`. For `@Value` support, you need to either additionally configure a `PropertySourcesPlaceholderConfigurer` or use `@SpringBootTest`, which auto-configures one for you.

TestPropertyValues

`TestPropertyValues` lets you quickly add properties to a `ConfigurableEnvironment` or `ConfigurableApplicationContext`. You can call it with `key=value` strings, as follows:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.boot.test.util.TestPropertyValues;
import org.springframework.mock.env.MockEnvironment;

import static org.assertj.core.api.Assertions.assertThat;

class MyEnvironmentTests {

    @Test
    void testPropertySources() {
        MockEnvironment environment = new MockEnvironment();
        TestPropertyValues.of("org=Spring", "name=Boot").applyTo(environment);
        assertThat(environment.getProperty("name")).isEqualTo("Boot");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.boot.test.util.TestPropertyValues
import org.springframework.mock.env.MockEnvironment

class MyEnvironmentTests {

    @Test
    fun testPropertySources() {
        val environment = MockEnvironment()
        TestPropertyValues.of("org=Spring", "name=Boot").applyTo(environment)
        assertThat(environment.getProperty("name")).isEqualTo("Boot")
    }

}
```

OutputCapture

`OutputCapture` is a JUnit `Extension` that you can use to capture `System.out` and `System.err` output. To use it, add `@ExtendWith(OutputCaptureExtension.class)` and inject `CapturedOutput` as an argument to your test class constructor or test method as follows:

Java

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.boot.test.system.CapturedOutput;
import org.springframework.boot.test.system.OutputCaptureExtension;

import static org.assertj.core.api.Assertions.assertThat;

@ExtendWith(OutputCaptureExtension.class)
class MyOutputCaptureTests {

    @Test
    void testName(CapturedOutput output) {
        System.out.println("Hello World!");
        assertThat(output).contains("World");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.extension.ExtendWith
import org.springframework.boot.test.system.CapturedOutput
import org.springframework.boot.test.system.OutputCaptureExtension

@ExtendWith(OutputCaptureExtension::class)
class MyOutputCaptureTests {

    @Test
    fun testName(output: CapturedOutput?) {
        println("Hello World!")
        assertThat(output).contains("World")
    }

}
```

TestRestTemplate

`TestRestTemplate` is a convenience alternative to Spring's `RestTemplate` that is useful in integration tests. You can get a vanilla template or one that sends Basic HTTP authentication (with a username and password). In either case, the template is fault tolerant. This means that it behaves in a test-friendly way by not throwing exceptions on 4xx and 5xx errors. Instead, such errors can be detected through the returned `ResponseEntity` and its status code.

TIP	Spring Framework 5.0 provides a new `WebTestClient` that works for WebFlux integration tests and both WebFlux and MVC end-to-end testing. It provides a fluent API for assertions, unlike `TestRestTemplate`.

It is recommended, but not mandatory, to use the Apache HTTP Client (version 5.1 or better). If you have that on your classpath, the `TestRestTemplate` responds by configuring the client appropriately. If you do use Apache's HTTP client, some additional test-friendly features are enabled:

- Redirects are not followed (so you can assert the response location).
- Cookies are ignored (so the template is stateless).

`TestRestTemplate` can be instantiated directly in your integration tests, as shown in the following example:

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.ResponseEntity;

import static org.assertj.core.api.Assertions.assertThat;

class MyTests {

    private final TestRestTemplate template = new TestRestTemplate();

    @Test
    void testRequest() {
        ResponseEntity<String> headers =
this.template.getForEntity("https://myhost.example.com/example", String.class);
        assertThat(headers.getHeaders().getLocation()).hasHost("other.example.com");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.boot.test.web.client.TestRestTemplate

class MyTests {

    private val template = TestRestTemplate()

    @Test
    fun testRequest() {
        val headers = template.getForEntity("https://myhost.example.com/example",
String::class.java)
        assertThat(headers.headers.location).hasHost("other.example.com")
    }

}
```

Alternatively, if you use the `@SpringBootTest` annotation with `WebEnvironment.RANDOM_PORT` or `WebEnvironment.DEFINED_PORT`, you can inject a fully configured `TestRestTemplate` and start using it. If necessary, additional customizations can be applied through the `RestTemplateBuilder` bean. Any URLs that do not specify a host and port automatically connect to the embedded server, as shown in the following example:

Java

```
import java.time.Duration;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.http.HttpHeaders;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MySpringBootTests {

    @Autowired
    private TestRestTemplate template;

    @Test
    void testRequest() {
        HttpHeaders headers = this.template.getForEntity("/example",
String.class).getHeaders();
        assertThat(headers.getLocation()).hasHost("other.example.com");
    }

    @TestConfiguration(proxyBeanMethods = false)
    static class RestTemplateBuilderConfiguration {

        @Bean
        RestTemplateBuilder restTemplateBuilder() {
            return new RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1))
                .setReadTimeout(Duration.ofSeconds(1));
        }

    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.test.web.client.TestRestTemplate
import org.springframework.boot.web.client.RestTemplateBuilder
import org.springframework.context.annotation.Bean
import java.time.Duration

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MySpringBootTests(@Autowired val template: TestRestTemplate) {

    @Test
    fun testRequest() {
        val headers = template.getForEntity("/example", String::class.java).headers
        assertThat(headers.location).hasHost("other.example.com")
    }

    @TestConfiguration(proxyBeanMethods = false)
    internal class RestTemplateBuilderConfiguration {

        @Bean
        fun restTemplateBuilder(): RestTemplateBuilder {
            return RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1))
                .setReadTimeout(Duration.ofSeconds(1))
        }

    }

}
```

7.10. Docker Compose Support

Docker Compose is a popular technology that can be used to define and manage multiple containers for services that your application needs. A `compose.yml` file is typically created next to your application which defines and configures service containers.

A typical workflow with Docker Compose is to run `docker compose up`, work on your application with it connecting to started services, then run `docker compose down` when you are finished.

The `spring-boot-docker-compose` module can be included in a project to provide support for working with containers using Docker Compose. Add the module dependency to your build, as shown in the following listings for Maven and Gradle:

Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-docker-compose</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

Gradle

```
dependencies {
    developmentOnly("org.springframework.boot:spring-boot-docker-compose")
}
```

NOTE	The `docker compose` or `docker-compose` CLI application needs to be on your path in order for Spring Boot's support to work correctly.

When this module is included as a dependency Spring Boot will do the following:

- Search for a `compose.yml` and other common compose filenames in your application directory
- Call `docker compose up` with the discovered `compose.yml`
- Create service connection beans for each supported container
- Call `docker compose stop` when the application is shutdown

If the Docker Compose services are already running when starting the application, Spring Boot will only create the service connection beans for each supported container. It will not call `docker compose up` again and it will not call `docker compose stop` when the application is shutdown.

NOTE	By default, Spring Boot's Docker Compose support is disabled when running tests. To enable it, set `spring.docker.compose.skip.in-tests` to `false`.

7.10.1. Service Connections

A service connection is a connection to any remote service. Spring Boot's auto-configuration can consume the details of a service connection and use them to establish a connection to a remote service. When doing so, the connection details take precedence over any connection-related configuration properties.

When using Spring Boot's Docker Compose support, service connections are established to the port mapped by the container.

NOTE	Docker compose is usually used in such a way that the ports inside the container are mapped to ephemeral ports on your computer. For example, a Postgres server may run inside the container using port 5432 but be mapped to a totally different port locally. The service connection will always discover and use the locally mapped port.

Service connections are established by using the image name of the container. The following service connections are currently supported:

Connection Details	Matched on
`CassandraConnectionDetails`	Containers named "cassandra"
`ElasticsearchConnectionDetails`	Containers named "elasticsearch"
`JdbcConnectionDetails`	Containers named "gvenzl/oracle-xe", "mariadb", "mssql/server", "mysql", or "postgres"
`MongoConnectionDetails`	Containers named "mongo"
`R2dbcConnectionDetails`	Containers named "gvenzl/oracle-xe", "mariadb", "mssql/server", "mysql", or "postgres"
`RabbitConnectionDetails`	Containers named "rabbitmq"
`RedisConnectionDetails`	Containers named "redis"
`ZipkinConnectionDetails`	Containers named "openzipkin/zipkin".

7.10.2. Custom Images

Sometimes you may need to use your own version of an image to provide a service. You can use any custom image as long as it behaves in the same way as the standard image. Specifically, any environment variables that the standard image supports must also be used in your custom image.

If your image uses a different name, you can use a label in your `compose.yml` file so that Spring Boot can provide a service connection. Use a label named `org.springframework.boot.service-connection` to provide the service name.

For example:

```
services:
  redis:
    image: 'mycompany/mycustomredis:7.0'
    ports:
      - '6379'
    labels:
      org.springframework.boot.service-connection: redis
```

7.10.3. Skipping Specific Containers

If you have a container image defined in your `compose.yml` that you don't want connected to your

application you can use a label to ignore it. Any container with labeled with `org.springframework.boot.ignore` will be ignored by Spring Boot.

For example:

```
services:
  redis:
    image: 'redis:7.0'
    ports:
      - '6379'
    labels:
      org.springframework.boot.ignore: true
```

7.10.4. Using a Specific Compose File

If your compose file is not in the same directory as your application, or if it's named differently, you can use `spring.docker.compose.file` in your `application.properties` or `application.yaml` to point to a different file. Properties can be defined as an exact path or a path that's relative to your application.

For example:

Properties

```
spring.docker.compose.file=../my-compose.yml
```

Yaml

```
spring:
  docker:
    compose:
      file: "../my-compose.yml"
```

7.10.5. Waiting for Container Readiness

Containers started by Docker Compose may take some time to become fully ready. The recommended way of checking for readiness is to add a `healthcheck` section under the service definition in your `compose.yml` file.

Since it's not uncommon for `healthcheck` configuration to be omitted from `compose.yml` files, Spring Boot also checks directly for service readiness. By default, a container is considered ready when a TCP/IP connection can be established to its mapped port.

You can disable this on a per-container basis by adding a `org.springframework.boot.readiness-check.tcp.disable` label in your `compose.yml` file.

For example:

```
services:
  redis:
    image: 'redis:7.0'
    ports:
      - '6379'
    labels:
      org.springframework.boot.readiness-check.tcp.disable: true
```

You can also change timeout values in your `application.properties` or `application.yaml` file:

Properties

```
spring.docker.compose.readiness.tcp.connect-timeout=10s
spring.docker.compose.readiness.tcp.read-timeout=5s
```

Yaml

```
spring:
  docker:
    compose:
      readiness:
        tcp:
          connect-timeout: 10s
          read-timeout: 5s
```

The overall timeout can be configured using `spring.docker.compose.readiness.timeout`.

7.10.6. Controlling the Docker Compose Lifecycle

By default Spring Boot calls `docker compose up` when your application starts and `docker compose stop` when it's shut down. If you prefer to have different lifecycle management you can use the `spring.docker.compose.lifecycle-management` property.

The following values are supported:

- `none` - Do not start or stop Docker Compose
- `start-only` - Start Docker Compose when the application starts and leave it running
- `start-and-stop` - Start Docker Compose when the application starts and stop it when the JVM exits

In addition you can use the `spring.docker.compose.start.command` property to change whether `docker compose up` or `docker compose start` is used. The `spring.docker.compose.stop.command` allows you to configure if `docker compose down` or `docker compose stop` is used.

The following example shows how lifecycle management can be configured:

Properties

```
spring.docker.compose.lifecycle-management=start-and-stop
spring.docker.compose.start.command=start
spring.docker.compose.stop.command=down
spring.docker.compose.stop.timeout=1m
```

Yaml

```
spring:
  docker:
    compose:
      lifecycle-management: start-and-stop
      start:
        command: start
      stop:
        command: down
        timeout: 1m
```

7.10.7. Activating Docker Compose Profiles

Docker Compose profiles are similar to Spring profiles in that they let you adjust your Docker Compose configuration for specific environments. If you want to activate a specific Docker Compose profile you can use the `spring.docker.compose.profiles.active` property in your `application.properties` or `application.yaml` file:

Properties

```
spring.docker.compose.profiles.active=myprofile
```

Yaml

```
spring:
  docker:
    compose:
      profiles:
        active: "myprofile"
```

7.11. Testcontainers Support

As well as using Testcontainers for integration testing, it's also possible to use them at development time. The next sections will provide more details about that.

7.11.1. Using Testcontainers at Development Time

This approach allows developers to quickly start containers for the services that the application depends on, removing the need to manually provision things like database servers. Using

Testcontainers in this way provides functionality similar to Docker Compose, except that your container configuration is in Java rather than YAML.

To use Testcontainers at development time you need to launch your application using your "test" classpath rather than "main". This will allow you to access all declared test dependencies and give you a natural place to write your test configuration.

To create a test launchable version of your application you should create an "Application" class in the `src/test` directory. For example, if your main application is in `src/main/java/com/example/MyApplication.java`, you should create `src/test/java/com/example/TestMyApplication.java`

The `TestMyApplication` class can use the `SpringApplication.from(…)` method to launch the real application:

Java

```
import org.springframework.boot.SpringApplication;

public class TestMyApplication {

    public static void main(String[] args) {
        SpringApplication.from(MyApplication::main).run(args);
    }

}
```

Kotlin

```
import org.springframework.boot.fromApplication

fun main(args: Array<String>) {
    fromApplication<MyApplication>().run(*args)
}
```

You'll also need to define the `Container` instances that you want to start along with your application. To do this, you need to make sure that the `spring-boot-testcontainers` module has been added as a `test` dependency. Once that has been done, you can create a `@TestConfiguration` class that declares `@Bean` methods for the containers you want to start.

You can also annotate your `@Bean` methods with `@ServiceConnection` in order to create `ConnectionDetails` beans. See the service connections section for details of the supported technologies.

A typical Testcontainers configuration would look like this:

Java

```
import org.testcontainers.containers.Neo4jContainer;

import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;
import org.springframework.context.annotation.Bean;

@TestConfiguration(proxyBeanMethods = false)
public class MyContainersConfiguration {

    @Bean
    @ServiceConnection
    public Neo4jContainer<?> neo4jContainer() {
        return new Neo4jContainer<>("neo4j:5");
    }

}
```

Kotlin

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.service.connection.ServiceConnection
import org.springframework.context.annotation.Bean
import org.testcontainers.containers.Neo4jContainer

@TestConfiguration(proxyBeanMethods = false)
class MyContainersConfiguration {

    @Bean
    @ServiceConnection
    fun neo4jContainer(): Neo4jContainer<*> {
        return Neo4jContainer("neo4j:5")
    }

}
```

NOTE The lifecycle of `Container` beans is automatically managed by Spring Boot. Containers will be started and stopped automatically.

Once you have defined your test configuration, you can use the `with(…)` method to attach it to your test launcher:

Java

```
import org.springframework.boot.SpringApplication;

public class TestMyApplication {

    public static void main(String[] args) {

SpringApplication.from(MyApplication::main).with(MyContainersConfiguration.class).run(
args);
    }

}
```

Kotlin

```
import org.springframework.boot.fromApplication
import org.springframework.boot.with

fun main(args: Array<String>) {
    fromApplication<MyApplication>().with(MyContainersConfiguration::class).run(*args)
}
```

You can now launch `TestMyApplication` as you would any regular Java `main` method application to start your application and the containers that it needs to run.

TIP	You can use the Maven goal `spring-boot:test-run` or the Gradle task `bootTestRun` to do this from the command line.

Contributing Dynamic Properties at Development Time

If you want to contribute dynamic properties at development time from your `Container @Bean` methods, you can do so by injecting a `DynamicPropertyRegistry`. This works in a similar way to the `@DynamicPropertySource` annotation that you can use in your tests. It allows you to add properties that will become available once your container has started.

A typical configuration would look like this:

Java

```
import org.testcontainers.containers.MongoDBContainer;

import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.test.context.DynamicPropertyRegistry;

@TestConfiguration(proxyBeanMethods = false)
public class MyContainersConfiguration {

    @Bean
    public MongoDBContainer mongoDbContainer(DynamicPropertyRegistry properties) {
        MongoDBContainer container = new MongoDBContainer("mongo:5.0");
        properties.add("spring.data.mongodb.host", container::getHost);
        properties.add("spring.data.mongodb.port", container::getFirstMappedPort);
        return container;
    }

}
```

Kotlin

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.context.annotation.Bean;
import org.springframework.test.context.DynamicPropertyRegistry
import org.testcontainers.containers.MongoDBContainer

@TestConfiguration(proxyBeanMethods = false)
class MyContainersConfiguration {

    @Bean
    fun monogDbContainer(properties: DynamicPropertyRegistry): MongoDBContainer {
        var container = MongoDBContainer("mongo:5.0")
        properties.add("spring.data.mongodb.host", container::getHost);
        properties.add("spring.data.mongodb.port", container::getFirstMappedPort);
        return container
    }

}
```

NOTE	Using a `@ServiceConnection` is recommended whenever possible, however, dynamic properties can be a useful fallback for technologies that don't yet have `@ServiceConnection` support.

Importing Testcontainer Declaration Classes

A common pattern when using Testcontainers is to declare `Container` instances as static fields. Often these fields are defined directly on the test class. They can also be declared on a parent class or on

an interface that the test implements.

For example, the following `MyContainers` interface declares `mongo` and `neo4j` containers:

```
import org.testcontainers.containers.MongoDBContainer;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;

import org.springframework.boot.testcontainers.service.connection.ServiceConnection;

public interface MyContainers {

    @Container
    @ServiceConnection
    MongoDBContainer mongoContainer = new MongoDBContainer("mongo:5.0");

    @Container
    @ServiceConnection
    Neo4jContainer<?> neo4jContainer = new Neo4jContainer<>("neo4j:5");

}
```

If you already have containers defined in this way, or you just prefer this style, you can import these declaration classes rather than defining you containers as `@Bean` methods. To do so, add the `@ImportTestcontainers` annotation to your test configuration class:

Java

```
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.context.ImportTestcontainers;

@TestConfiguration(proxyBeanMethods = false)
@ImportTestcontainers(MyContainers.class)
public class MyContainersConfiguration {

}
```

Kotlin

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.context.ImportTestcontainers

@TestConfiguration(proxyBeanMethods = false)
@ImportTestcontainers(MyContainers::class)
class MyContainersConfiguration {

}
```

TIP	If you don't intend to use the service connections feature but want to use `@DynamicPropertySource` instead, remove the `@ServiceConnection` annotation from the `Container` fields. You can also add `@DynamicPropertySource` annotated methods to your declaration class.

Using DevTools with Testcontainers at Development Time

When using devtools, you can annotate beans and bean methods with `@RestartScope`. Such beans won't be recreated when the devtools restart the application. This is especially useful for Testcontainer `Container` beans, as they keep their state despite the application restart.

Java

```
import org.testcontainers.containers.MongoDBContainer;

import org.springframework.boot.devtools.restart.RestartScope;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;
import org.springframework.context.annotation.Bean;

@TestConfiguration(proxyBeanMethods = false)
public class MyContainersConfiguration {

    @Bean
    @RestartScope
    @ServiceConnection
    public MongoDBContainer mongoDbContainer() {
        return new MongoDBContainer("mongo:5.0");
    }

}
```

Kotlin

```
import org.springframework.boot.devtools.restart.RestartScope
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;
import org.springframework.context.annotation.Bean
import org.testcontainers.containers.MongoDBContainer

@TestConfiguration(proxyBeanMethods = false)
class MyContainersConfiguration {

    @Bean
    @RestartScope
    @ServiceConnection
    fun monogDbContainer(): MongoDBContainer {
        return MongoDBContainer("mongo:5.0")
    }

}
```

WARNING	If you're using Gradle and want to use this feature, you need to change the configuration of the `spring-boot-devtools` dependency from `developmentOnly` to `testImplementation`. With the default scope of `developmentOnly`, the `bootTestRun` task will not pick up changes in your code, as the devtools are not active.

7.12. Creating Your Own Auto-configuration

If you work in a company that develops shared libraries, or if you work on an open-source or commercial library, you might want to develop your own auto-configuration. Auto-configuration classes can be bundled in external jars and still be picked up by Spring Boot.

Auto-configuration can be associated to a "starter" that provides the auto-configuration code as well as the typical libraries that you would use with it. We first cover what you need to know to build your own auto-configuration and then we move on to the typical steps required to create a custom starter.

7.12.1. Understanding Auto-configured Beans

Classes that implement auto-configuration are annotated with `@AutoConfiguration`. This annotation itself is meta-annotated with `@Configuration`, making auto-configurations standard `@Configuration` classes. Additional `@Conditional` annotations are used to constrain when the auto-configuration should apply. Usually, auto-configuration classes use `@ConditionalOnClass` and `@ConditionalOnMissingBean` annotations. This ensures that auto-configuration applies only when relevant classes are found and when you have not declared your own `@Configuration`.

You can browse the source code of `spring-boot-autoconfigure` to see the `@AutoConfiguration` classes that Spring provides (see the `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file).

7.12.2. Locating Auto-configuration Candidates

Spring Boot checks for the presence of a `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file within your published jar. The file should list your configuration classes, with one class name per line, as shown in the following example:

```
com.mycorp.libx.autoconfigure.LibXAutoConfiguration
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

TIP You can add comments to the imports file using the `#` character.

NOTE Auto-configurations must be loaded *only* by being named in the imports file. Make sure that they are defined in a specific package space and that they are never the target of component scanning. Furthermore, auto-configuration classes should not enable component scanning to find additional components. Specific `@Import` annotations should be used instead.

If your configuration needs to be applied in a specific order, you can use the `before`, `beforeName`, `after` and `afterName` attributes on the `@AutoConfiguration` annotation or the dedicated `@AutoConfigureBefore` and `@AutoConfigureAfter` annotations. For example, if you provide web-specific configuration, your class may need to be applied after `WebMvcAutoConfiguration`.

If you want to order certain auto-configurations that should not have any direct knowledge of each other, you can also use `@AutoConfigureOrder`. That annotation has the same semantic as the regular `@Order` annotation but provides a dedicated order for auto-configuration classes.

As with standard `@Configuration` classes, the order in which auto-configuration classes are applied only affects the order in which their beans are defined. The order in which those beans are subsequently created is unaffected and is determined by each bean's dependencies and any `@DependsOn` relationships.

7.12.3. Condition Annotations

You almost always want to include one or more `@Conditional` annotations on your auto-configuration class. The `@ConditionalOnMissingBean` annotation is one common example that is used to allow developers to override auto-configuration if they are not happy with your defaults.

Spring Boot includes a number of `@Conditional` annotations that you can reuse in your own code by annotating `@Configuration` classes or individual `@Bean` methods. These annotations include:

- Class Conditions
- Bean Conditions
- Property Conditions
- Resource Conditions
- Web Application Conditions

- SpEL Expression Conditions

Class Conditions

The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations let `@Configuration` classes be included based on the presence or absence of specific classes. Due to the fact that annotation metadata is parsed by using ASM, you can use the `value` attribute to refer to the real class, even though that class might not actually appear on the running application classpath. You can also use the `name` attribute if you prefer to specify the class name by using a `String` value.

This mechanism does not apply the same way to `@Bean` methods where typically the return type is the target of the condition: before the condition on the method applies, the JVM will have loaded the class and potentially processed method references which will fail if the class is not present.

To handle this scenario, a separate `@Configuration` class can be used to isolate the condition, as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.AutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@AutoConfiguration
// Some conditions ...
public class MyAutoConfiguration {

    // Auto-configured beans ...

    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(SomeService.class)
    public static class SomeServiceConfiguration {

        @Bean
        @ConditionalOnMissingBean
        public SomeService someService() {
            return new SomeService();
        }

    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
// Some conditions ...
class MyAutoConfiguration {

    // Auto-configured beans ...
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(SomeService::class)
    class SomeServiceConfiguration {

        @Bean
        @ConditionalOnMissingBean
        fun someService(): SomeService {
            return SomeService()
        }

    }

}
```

TIP If you use `@ConditionalOnClass` or `@ConditionalOnMissingClass` as a part of a meta-annotation to compose your own composed annotations, you must use `name` as referring to the class in such a case is not handled.

Bean Conditions

The `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations let a bean be included based on the presence or absence of specific beans. You can use the `value` attribute to specify beans by type or `name` to specify beans by name. The `search` attribute lets you limit the `ApplicationContext` hierarchy that should be considered when searching for beans.

When placed on a `@Bean` method, the target type defaults to the return type of the method, as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.AutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;

@AutoConfiguration
public class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public SomeService someService() {
        return new SomeService();
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    fun someService(): SomeService {
        return SomeService()
    }

}
```

In the preceding example, the `someService` bean is going to be created if no bean of type `SomeService` is already contained in the `ApplicationContext`.

TIP You need to be very careful about the order in which bean definitions are added, as these conditions are evaluated based on what has been processed so far. For this reason, we recommend using only `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations on auto-configuration classes (since these are guaranteed to load after any user-defined bean definitions have been added).

NOTE `@ConditionalOnBean` and `@ConditionalOnMissingBean` do not prevent `@Configuration` classes from being created. The only difference between using these conditions at the class level and marking each contained `@Bean` method with the annotation is that the former prevents registration of the `@Configuration` class as a bean if the condition does not match.

TIP	When declaring a `@Bean` method, provide as much type information as possible in the method's return type. For example, if your bean's concrete class implements an interface the bean method's return type should be the concrete class and not the interface. Providing as much type information as possible in `@Bean` methods is particularly important when using bean conditions as their evaluation can only rely upon to type information that is available in the method signature.

Property Conditions

The `@ConditionalOnProperty` annotation lets configuration be included based on a Spring Environment property. Use the `prefix` and `name` attributes to specify the property that should be checked. By default, any property that exists and is not equal to `false` is matched. You can also create more advanced checks by using the `havingValue` and `matchIfMissing` attributes.

Resource Conditions

The `@ConditionalOnResource` annotation lets configuration be included only when a specific resource is present. Resources can be specified by using the usual Spring conventions, as shown in the following example: `file:/home/user/test.dat`.

Web Application Conditions

The `@ConditionalOnWebApplication` and `@ConditionalOnNotWebApplication` annotations let configuration be included depending on whether the application is a web application. A servlet-based web application is any application that uses a Spring `WebApplicationContext`, defines a `session` scope, or has a `ConfigurableWebEnvironment`. A reactive web application is any application that uses a `ReactiveWebApplicationContext`, or has a `ConfigurableReactiveWebEnvironment`.

The `@ConditionalOnWarDeployment` and `@ConditionalOnNotWarDeployment` annotations let configuration be included depending on whether the application is a traditional WAR application that is deployed to a servlet container. This condition will not match for applications that are run with an embedded web server.

SpEL Expression Conditions

The `@ConditionalOnExpression` annotation lets configuration be included based on the result of a SpEL expression.

NOTE	Referencing a bean in the expression will cause that bean to be initialized very early in context refresh processing. As a result, the bean won't be eligible for post-processing (such as configuration properties binding) and its state may be incomplete.

7.12.4. Testing your Auto-configuration

An auto-configuration can be affected by many factors: user configuration (`@Bean` definition and `Environment` customization), condition evaluation (presence of a particular library), and others. Concretely, each test should create a well defined `ApplicationContext` that represents a combination of those customizations. `ApplicationContextRunner` provides a great way to achieve that.

WARNING	`ApplicationContextRunner` doesn't work when running the tests in a native image.

`ApplicationContextRunner` is usually defined as a field of the test class to gather the base, common configuration. The following example makes sure that `MyServiceAutoConfiguration` is always invoked:

Java

```
private final ApplicationContextRunner contextRunner = new ApplicationContextRunner()
    .withConfiguration(AutoConfigurations.of(MyServiceAutoConfiguration.class));
```

Kotlin

```
val contextRunner = ApplicationContextRunner()
    .withConfiguration(AutoConfigurations.of(MyServiceAutoConfiguration::class.java))
```

TIP	If multiple auto-configurations have to be defined, there is no need to order their declarations as they are invoked in the exact same order as when running the application.

Each test can use the runner to represent a particular use case. For instance, the sample below invokes a user configuration (`UserConfiguration`) and checks that the auto-configuration backs off properly. Invoking `run` provides a callback context that can be used with `AssertJ`.

Java

```
@Test
void defaultServiceBacksOff() {
    this.contextRunner.withUserConfiguration(UserConfiguration.class).run((context) ->
{
        assertThat(context).hasSingleBean(MyService.class);

assertThat(context).getBean("myCustomService").isSameAs(context.getBean(MyService.clas
s));
    });
}

@Configuration(proxyBeanMethods = false)
static class UserConfiguration {

    @Bean
    MyService myCustomService() {
        return new MyService("mine");
    }

}
```

Kotlin

```
@Test
fun defaultServiceBacksOff() {
    contextRunner.withUserConfiguration(UserConfiguration::class.java)
        .run { context: AssertableApplicationContext ->
            assertThat(context).hasSingleBean(MyService::class.java)
            assertThat(context).getBean("myCustomService")
                .isSameAs(context.getBean(MyService::class.java))
        }
}

@Configuration(proxyBeanMethods = false)
internal class UserConfiguration {

    @Bean
    fun myCustomService(): MyService {
        return MyService("mine")
    }

}
```

It is also possible to easily customize the `Environment`, as shown in the following example:

Java

```
@Test
void serviceNameCanBeConfigured() {
    this.contextRunner.withPropertyValues("user.name=test123").run((context) -> {
        assertThat(context).hasSingleBean(MyService.class);
        assertThat(context.getBean(MyService.class).getName()).isEqualTo("test123");
    });
}
```

Kotlin

```
@Test
fun serviceNameCanBeConfigured() {
    contextRunner.withPropertyValues("user.name=test123").run { context:
AssertableApplicationContext ->
        assertThat(context).hasSingleBean(MyService::class.java)
        assertThat(context.getBean(MyService::class.java).name).isEqualTo("test123")
    }
}
```

The runner can also be used to display the `ConditionEvaluationReport`. The report can be printed at `INFO` or `DEBUG` level. The following example shows how to use the `ConditionEvaluationReportLoggingListener` to print the report in auto-configuration tests.

Java

```
import org.junit.jupiter.api.Test;

import
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListene
r;
import org.springframework.boot.logging.LogLevel;
import org.springframework.boot.test.context.runner.ApplicationContextRunner;

class MyConditionEvaluationReportingTests {

    @Test
    void autoConfigTest() {
        new ApplicationContextRunner()

.withInitializer(ConditionEvaluationReportLoggingListener.forLogLevel(LogLevel.INFO))
            .run((context) -> {
                // Test something...
            });
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListene
r
import org.springframework.boot.logging.LogLevel
import org.springframework.boot.test.context.assertj.AssertableApplicationContext
import org.springframework.boot.test.context.runner.ApplicationContextRunner

class MyConditionEvaluationReportingTests {

    @Test
    fun autoConfigTest() {
        ApplicationContextRunner()

.withInitializer(ConditionEvaluationReportLoggingListener.forLogLevel(LogLevel.INFO))
            .run { context: AssertableApplicationContext? -> }
    }

}
```

Simulating a Web Context

If you need to test an auto-configuration that only operates in a servlet or reactive web application context, use the `WebApplicationContextRunner` or `ReactiveWebApplicationContextRunner` respectively.

Overriding the Classpath

It is also possible to test what happens when a particular class and/or package is not present at runtime. Spring Boot ships with a `FilteredClassLoader` that can easily be used by the runner. In the following example, we assert that if `MyService` is not present, the auto-configuration is properly disabled:

Java

```
@Test
void serviceIsIgnoredIfLibraryIsNotPresent() {
    this.contextRunner.withClassLoader(new FilteredClassLoader(MyService.class))
        .run((context) -> assertThat(context).doesNotHaveBean("myService"));
}
```

Kotlin

```
@Test
fun serviceIsIgnoredIfLibraryIsNotPresent() {
    contextRunner.withClassLoader(FilteredClassLoader(MyService::class.java))
        .run { context: AssertableApplicationContext? ->
            assertThat(context).doesNotHaveBean("myService")
        }
}
```

7.12.5. Creating Your Own Starter

A typical Spring Boot starter contains code to auto-configure and customize the infrastructure of a given technology, let's call that "acme". To make it easily extensible, a number of configuration keys in a dedicated namespace can be exposed to the environment. Finally, a single "starter" dependency is provided to help users get started as easily as possible.

Concretely, a custom starter can contain the following:

- The `autoconfigure` module that contains the auto-configuration code for "acme".
- The `starter` module that provides a dependency to the `autoconfigure` module as well as "acme" and any additional dependencies that are typically useful. In a nutshell, adding the starter should provide everything needed to start using that library.

This separation in two modules is in no way necessary. If "acme" has several flavors, options or optional features, then it is better to separate the auto-configuration as you can clearly express the fact some features are optional. Besides, you have the ability to craft a starter that provides an opinion about those optional dependencies. At the same time, others can rely only on the `autoconfigure` module and craft their own starter with different opinions.

If the auto-configuration is relatively straightforward and does not have optional features, merging the two modules in the starter is definitely an option.

Naming

You should make sure to provide a proper namespace for your starter. Do not start your module names with `spring-boot`, even if you use a different Maven `groupId`. We may offer official support for the thing you auto-configure in the future.

As a rule of thumb, you should name a combined module after the starter. For example, assume that you are creating a starter for "acme" and that you name the auto-configure module `acme-spring-boot` and the starter `acme-spring-boot-starter`. If you only have one module that combines the two, name it `acme-spring-boot-starter`.

Configuration keys

If your starter provides configuration keys, use a unique namespace for them. In particular, do not include your keys in the namespaces that Spring Boot uses (such as `server`, `management`, `spring`, and so on). If you use the same namespace, we may modify these namespaces in the future in ways that break your modules. As a rule of thumb, prefix all your keys with a namespace that you own (for example `acme`).

Make sure that configuration keys are documented by adding field javadoc for each property, as shown in the following example:

Java

```
import java.time.Duration;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("acme")
public class AcmeProperties {

    /**
     * Whether to check the location of acme resources.
     */
    private boolean checkLocation = true;

    /**
     * Timeout for establishing a connection to the acme server.
     */
    private Duration loginTimeout = Duration.ofSeconds(3);

    public boolean isCheckLocation() {
        return this.checkLocation;
    }

    public void setCheckLocation(boolean checkLocation) {
        this.checkLocation = checkLocation;
    }

    public Duration getLoginTimeout() {
        return this.loginTimeout;
    }

    public void setLoginTimeout(Duration loginTimeout) {
        this.loginTimeout = loginTimeout;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import java.time.Duration

@ConfigurationProperties("acme")
class AcmeProperties(

    /**
     * Whether to check the location of acme resources.
     */
    var isCheckLocation: Boolean = true,

    /**
     * Timeout for establishing a connection to the acme server.
     */
    var loginTimeout:Duration = Duration.ofSeconds(3))
```

NOTE	You should only use plain text with `@ConfigurationProperties` field Javadoc, since they are not processed before being added to the JSON.

Here are some rules we follow internally to make sure descriptions are consistent:

- Do not start the description by "The" or "A".
- For `boolean` types, start the description with "Whether" or "Enable".
- For collection-based types, start the description with "Comma-separated list"
- Use `java.time.Duration` rather than `long` and describe the default unit if it differs from milliseconds, such as "If a duration suffix is not specified, seconds will be used".
- Do not provide the default value in the description unless it has to be determined at runtime.

Make sure to trigger meta-data generation so that IDE assistance is available for your keys as well. You may want to review the generated metadata (`META-INF/spring-configuration-metadata.json`) to make sure your keys are properly documented. Using your own starter in a compatible IDE is also a good idea to validate that quality of the metadata.

The “autoconfigure” Module

The `autoconfigure` module contains everything that is necessary to get started with the library. It may also contain configuration key definitions (such as `@ConfigurationProperties`) and any callback interface that can be used to further customize how the components are initialized.

TIP	You should mark the dependencies to the library as optional so that you can include the `autoconfigure` module in your projects more easily. If you do it that way, the library is not provided and, by default, Spring Boot backs off.

Spring Boot uses an annotation processor to collect the conditions on auto-configurations in a metadata file (`META-INF/spring-autoconfigure-metadata.properties`). If that file is present, it is used

to eagerly filter auto-configurations that do not match, which will improve startup time.

When building with Maven, it is recommended to add the following dependency in a module that contains auto-configurations:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure-processor</artifactId>
    <optional>true</optional>
</dependency>
```

If you have defined auto-configurations directly in your application, make sure to configure the `spring-boot-maven-plugin` to prevent the `repackage` goal from adding the dependency into the fat jar:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.springframework.boot</groupId>
                            <artifactId>spring-boot-autoconfigure-
processor</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

With Gradle, the dependency should be declared in the `annotationProcessor` configuration, as shown in the following example:

```
dependencies {
    annotationProcessor "org.springframework.boot:spring-boot-autoconfigure-processor"
}
```

Starter Module

The starter is really an empty jar. Its only purpose is to provide the necessary dependencies to work with the library. You can think of it as an opinionated view of what is required to get started.

Do not make assumptions about the project in which your starter is added. If the library you are

auto-configuring typically requires other starters, mention them as well. Providing a proper set of *default* dependencies may be hard if the number of optional dependencies is high, as you should avoid including dependencies that are unnecessary for a typical usage of the library. In other words, you should not include optional dependencies.

NOTE Either way, your starter must reference the core Spring Boot starter (`spring-boot-starter`) directly or indirectly (there is no need to add it if your starter relies on another starter). If a project is created with only your custom starter, Spring Boot's core features will be honoured by the presence of the core starter.

7.13. Kotlin Support

Kotlin is a statically-typed language targeting the JVM (and other platforms) which allows writing concise and elegant code while providing interoperability with existing libraries written in Java.

Spring Boot provides Kotlin support by leveraging the support in other Spring projects such as Spring Framework, Spring Data, and Reactor. See the Spring Framework Kotlin support documentation for more information.

The easiest way to start with Spring Boot and Kotlin is to follow this comprehensive tutorial. You can create new Kotlin projects by using start.spring.io. Feel free to join the #spring channel of Kotlin Slack or ask a question with the `spring` and `kotlin` tags on Stack Overflow if you need support.

7.13.1. Requirements

Spring Boot requires at least Kotlin 1.7.x and manages a suitable Kotlin version through dependency management. To use Kotlin, `org.jetbrains.kotlin:kotlin-stdlib` and `org.jetbrains.kotlin:kotlin-reflect` must be present on the classpath. The `kotlin-stdlib` variants `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` can also be used.

Since Kotlin classes are final by default, you are likely to want to configure kotlin-spring plugin in order to automatically open Spring-annotated classes so that they can be proxied.

Jackson's Kotlin module is required for serializing / deserializing JSON data in Kotlin. It is automatically registered when found on the classpath. A warning message is logged if Jackson and Kotlin are present but the Jackson Kotlin module is not.

TIP These dependencies and plugins are provided by default if one bootstraps a Kotlin project on start.spring.io.

7.13.2. Null-safety

One of Kotlin's key features is null-safety. It deals with `null` values at compile time rather than deferring the problem to runtime and encountering a `NullPointerException`. This helps to eliminate a common source of bugs without paying the cost of wrappers like `Optional`. Kotlin also allows using functional constructs with nullable values as described in this comprehensive guide to null-safety in Kotlin.

Although Java does not allow one to express null-safety in its type system, Spring Framework, Spring Data, and Reactor now provide null-safety of their API through tooling-friendly annotations. By default, types from Java APIs used in Kotlin are recognized as platform types for which null-checks are relaxed. Kotlin's support for JSR 305 annotations combined with nullability annotations provide null-safety for the related Spring API in Kotlin.

The JSR 305 checks can be configured by adding the `-Xjsr305` compiler flag with the following options: `-Xjsr305={strict|warn|ignore}`. The default behavior is the same as `-Xjsr305=warn`. The `strict` value is required to have null-safety taken in account in Kotlin types inferred from Spring API but should be used with the knowledge that Spring API nullability declaration could evolve even between minor releases and more checks may be added in the future).

WARNING	Generic type arguments, varargs and array elements nullability are not yet supported. See SPR-15942 for up-to-date information. Also be aware that Spring Boot's own API is not yet annotated.

7.13.3. Kotlin API

runApplication

Spring Boot provides an idiomatic way to run an application with `runApplication<MyApplication>(*args)` as shown in the following example:

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

This is a drop-in replacement for `SpringApplication.run(MyApplication::class.java, *args)`. It also allows customization of the application as shown in the following example:

```
runApplication<MyApplication>(*args) {
    setBannerMode(OFF)
}
```

Extensions

Kotlin extensions provide the ability to extend existing classes with additional functionality. The Spring Boot Kotlin API makes use of these extensions to add new Kotlin specific conveniences to existing APIs.

`TestRestTemplate` extensions, similar to those provided by Spring Framework for `RestOperations` in

Spring Framework, are provided. Among other things, the extensions make it possible to take advantage of Kotlin reified type parameters.

7.13.4. Dependency management

In order to avoid mixing different versions of Kotlin dependencies on the classpath, Spring Boot imports the Kotlin BOM.

With Maven, the Kotlin version can be customized by setting the `kotlin.version` property and plugin management is provided for `kotlin-maven-plugin`. With Gradle, the Spring Boot plugin automatically aligns the `kotlin.version` with the version of the Kotlin plugin.

Spring Boot also manages the version of Coroutines dependencies by importing the Kotlin Coroutines BOM. The version can be customized by setting the `kotlin-coroutines.version` property.

TIP `org.jetbrains.kotlinx:kotlinx-coroutines-reactor` dependency is provided by default if one bootstraps a Kotlin project with at least one reactive dependency on start.spring.io.

7.13.5. @ConfigurationProperties

`@ConfigurationProperties` when used in combination with constructor binding supports classes with immutable `val` properties as shown in the following example:

```
@ConfigurationProperties("example.kotlin")
data class KotlinExampleProperties(
        val name: String,
        val description: String,
        val myService: MyService) {

    data class MyService(
            val apiToken: String,
            val uri: URI
    )
}
```

TIP To generate your own metadata using the annotation processor, `kapt` should be configured with the `spring-boot-configuration-processor` dependency. Note that some features (such as detecting the default value or deprecated items) are not working due to limitations in the model kapt provides.

7.13.6. Testing

While it is possible to use JUnit 4 to test Kotlin code, JUnit 5 is provided by default and is recommended. JUnit 5 enables a test class to be instantiated once and reused for all of the class's tests. This makes it possible to use `@BeforeAll` and `@AfterAll` annotations on non-static methods, which is a good fit for Kotlin.

To mock Kotlin classes, MockK is recommended. If you need the `MockK` equivalent of the Mockito specific `@MockBean` and `@SpyBean` annotations, you can use SpringMockK which provides similar `@MockkBean` and `@SpykBean` annotations.

7.13.7. Resources

Further reading

- Kotlin language reference
- Kotlin Slack (with a dedicated #spring channel)
- Stack Overflow with `spring` and `kotlin` tags
- Try Kotlin in your browser
- Kotlin blog
- Awesome Kotlin
- Tutorial: building web applications with Spring Boot and Kotlin
- Developing Spring Boot applications with Kotlin
- A Geospatial Messenger with Kotlin, Spring Boot and PostgreSQL
- Introducing Kotlin support in Spring Framework 5.0
- Spring Framework 5 Kotlin APIs, the functional way

Examples

- spring-boot-kotlin-demo: regular Spring Boot + Spring Data JPA project
- mixit: Spring Boot 2 + WebFlux + Reactive Spring Data MongoDB
- spring-kotlin-fullstack: WebFlux Kotlin fullstack example with Kotlin2js for frontend instead of JavaScript or TypeScript
- spring-petclinic-kotlin: Kotlin version of the Spring PetClinic Sample Application
- spring-kotlin-deepdive: a step by step migration for Boot 1.0 + Java to Boot 2.0 + Kotlin
- spring-boot-coroutines-demo: Coroutines sample project

7.14. SSL

Spring Boot provides the ability to configure SSL trust material that can be applied to several types of connections in order to support secure communications. Configuration properties with the prefix `spring.ssl.bundle` can be used to specify named sets of trust material and associated information.

7.14.1. Configuring SSL With Java KeyStore Files

Configuration properties with the prefix `spring.ssl.bundle.jks` can be used to configure bundles of trust material created with the Java `keytool` utility and stored in Java KeyStore files in the JKS or PKCS12 format. Each bundle has a user-provided name that can be used to reference the bundle.

When used to secure an embedded web server, a `keystore` is typically configured with a Java

KeyStore containing a certificate and private key as shown in this example:

Properties

```
spring.ssl.bundle.jks.mybundle.key.alias=application
spring.ssl.bundle.jks.mybundle.keystore.location=classpath:application.p12
spring.ssl.bundle.jks.mybundle.keystore.password=secret
spring.ssl.bundle.jks.mybundle.keystore.type=PKCS12
```

Yaml

```
spring:
  ssl:
    bundle:
      jks:
        mybundle:
          key:
            alias: "application"
          keystore:
            location: "classpath:application.p12"
            password: "secret"
            type: "PKCS12"
```

When used to secure a client-side connection, a `truststore` is typically configured with a Java KeyStore containing the server certificate as shown in this example:

Properties

```
spring.ssl.bundle.jks.mybundle.truststore.location=classpath:server.p12
spring.ssl.bundle.jks.mybundle.truststore.password=secret
```

Yaml

```
spring:
  ssl:
    bundle:
      jks:
        mybundle:
          truststore:
            location: "classpath:server.p12"
            password: "secret"
```

See JksSslBundleProperties for the full set of supported properties.

7.14.2. Configuring SSL With PEM-encoded Certificates

Configuration properties with the prefix `spring.ssl.bundle.pem` can be used to configure bundles of trust material in the form of PEM-encoded text. Each bundle has a user-provided name that can be

used to reference the bundle.

When used to secure an embedded web server, a `keystore` is typically configured with a certificate and private key as shown in this example:

Properties

```
spring.ssl.bundle.pem.mybundle.keystore.certificate=classpath:application.crt
spring.ssl.bundle.pem.mybundle.keystore.private-key=classpath:application.key
```

Yaml

```
spring:
  ssl:
    bundle:
      pem:
        mybundle:
          keystore:
            certificate: "classpath:application.crt"
            private-key: "classpath:application.key"
```

When used to secure a client-side connection, a `truststore` is typically configured with the server certificate as shown in this example:

Properties

```
spring.ssl.bundle.pem.mybundle.truststore.certificate=classpath:server.crt
```

Yaml

```
spring:
  ssl:
    bundle:
      pem:
        mybundle:
          truststore:
            certificate: "classpath:server.crt"
```

PEM content can be used directly for both the `certificate` and `private-key` properties. If the property values contains `BEGIN` and `END` markers then they will be treated as PEM content rather than a resource location.

The following example shows how a truststore certificate can be defined:

Properties

```
spring.ssl.bundle.pem.mybundle.truststore.certificate=\
-----BEGIN CERTIFICATE-----\n\
MIID1zCCAr+gAwIBAgIUNM5QQv8IzVQsgSmmdPQNaqyzWs4wDQYJKoZIhvcNAQEL\n\
BQAwezELMAkGA1UEBhMCWFgxEjAQBgNVBAgMCVN0YXRlTmFtZTERMA8GA1UEBwwI\n\
...\n\
V0IJjcmYjEZbTvpjFKznvaFiOUv+8L7jHQ1/Yf+9c3C8gSjdUfv88m17pqYXd+Ds\n\
HEmfmNNjht130UyjNCITmLVXyy5p35vWmdf95U3uEbJSnNVtXH8qRmN9oK9mUpDb\n\
ngX6JBJI7fw7tXoqWSLHNiBODM88fUlQSho8\n\
-----END CERTIFICATE-----\n
```

Yaml

```
spring:
  ssl:
    bundle:
      pem:
        mybundle:
          truststore:
            certificate: |
              -----BEGIN CERTIFICATE-----

MIID1zCCAr+gAwIBAgIUNM5QQv8IzVQsgSmmdPQNaqyzWs4wDQYJKoZIhvcNAQEL

BQAwezELMAkGA1UEBhMCWFgxEjAQBgNVBAgMCVN0YXRlTmFtZTERMA8GA1UEBwwI
              ...

V0IJjcmYjEZbTvpjFKznvaFiOUv+8L7jHQ1/Yf+9c3C8gSjdUfv88m17pqYXd+Ds

HEmfmNNjht130UyjNCITmLVXyy5p35vWmdf95U3uEbJSnNVtXH8qRmN9oK9mUpDb
              ngX6JBJI7fw7tXoqWSLHNiBODM88fUlQSho8
              -----END CERTIFICATE-----
```

TIP

See PemSslBundleProperties for the full set of supported properties.

7.14.3. Applying SSL Bundles

Once configured using properties, SSL bundles can be referred to by name in configuration properties for various types of connections that are auto-configured by Spring Boot. See the sections on embedded web servers, data technologies, and REST clients for further information.

7.14.4. Using SSL Bundles

Spring Boot auto-configures a bean of type `SslBundles` that provides access to each of the named bundles configured using the `spring.ssl.bundle` properties.

An `SslBundle` can be retrieved from the auto-configured `SslBundles` bean and used to create objects that are used to configure SSL connectivity in client libraries. The `SslBundle` provides a layered approach of obtaining these SSL objects:

- `getStores()` provides access to the key store and trust store `java.security.KeyStore` instances as well as any required key store password.
- `getManagers()` provides access to the `java.net.ssl.KeyManagerFactory` and `java.net.ssl.TrustManagerFactory` instances as well as the `java.net.ssl.KeyManager` and `java.net.ssl.TrustManager` arrays that they create.
- `createSslContext()` provides a convenient way to obtain a new `java.net.ssl.SSLContext` instance.

In addition, the `SslBundle` provides details about the key being used, the protocol to use and any option that should be applied to the SSL engine.

The following example shows retrieving an `SslBundle` and using it to create an `SSLContext`:

Java

```
import javax.net.ssl.SSLContext;

import org.springframework.boot.ssl.SslBundle;
import org.springframework.boot.ssl.SslBundles;
import org.springframework.stereotype.Component;

@Component
public class MyComponent {

    public MyComponent(SslBundles sslBundles) {
        SslBundle sslBundle = sslBundles.getBundle("mybundle");
        SSLContext sslContext = sslBundle.createSslContext();
        // do something with the created sslContext
    }

}
```

Kotlin

```
import org.springframework.boot.ssl.SslBundles
import org.springframework.stereotype.Component

@Component
class MyComponent(sslBundles: SslBundles) {

    init {
        val sslBundle = sslBundles.getBundle("mybundle")
        val sslContext = sslBundle.createSslContext()
        // do something with the created sslContext
    }

}
```

7.15. What to Read Next

If you want to learn more about any of the classes discussed in this section, see the Spring Boot API documentation or you can browse the source code directly. If you have specific questions, see the how-to section.

If you are comfortable with Spring Boot's core features, you can continue on and read about production-ready features.

Chapter 8. Web

Spring Boot is well suited for web application development. You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the `spring-boot-starter-web` module to get up and running quickly. You can also choose to build reactive web applications by using the `spring-boot-starter-webflux` module.

If you have not yet developed a Spring Boot web application, you can follow the "Hello World!" example in the *Getting started* section.

8.1. Servlet Web Applications

If you want to build servlet-based web applications, you can take advantage of Spring Boot's auto-configuration for Spring MVC or Jersey.

8.1.1. The "Spring Web MVC Framework"

The Spring Web MVC framework (often referred to as "Spring MVC") is a rich "model view controller" web framework. Spring MVC lets you create special `@Controller` or `@RestController` beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using `@RequestMapping` annotations.

The following code shows a typical `@RestController` that serves JSON data:

Java

```
import java.util.List;

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class MyRestController {

    private final UserRepository userRepository;

    private final CustomerRepository customerRepository;

    public MyRestController(UserRepository userRepository, CustomerRepository
customerRepository) {
        this.userRepository = userRepository;
        this.customerRepository = customerRepository;
    }

    @GetMapping("/{userId}")
    public User getUser(@PathVariable Long userId) {
        return this.userRepository.findById(userId).get();
    }

    @GetMapping("/{userId}/customers")
    public List<Customer> getUserCustomers(@PathVariable Long userId) {
        return
this.userRepository.findById(userId).map(this.customerRepository::findByUser).get();
    }

    @DeleteMapping("/{userId}")
    public void deleteUser(@PathVariable Long userId) {
        this.userRepository.deleteById(userId);
    }

}
```

Kotlin

```
import org.springframework.web.bind.annotation.DeleteMapping
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController


@RestController
@RequestMapping("/users")
class MyRestController(private val userRepository: UserRepository, private val
customerRepository: CustomerRepository) {

    @GetMapping("/{userId}")
    fun getUser(@PathVariable userId: Long): User {
        return userRepository.findById(userId).get()
    }

    @GetMapping("/{userId}/customers")
    fun getUserCustomers(@PathVariable userId: Long): List<Customer> {
        return
userRepository.findById(userId).map(customerRepository::findByUser).get()
    }

    @DeleteMapping("/{userId}")
    fun deleteUser(@PathVariable userId: Long) {
        userRepository.deleteById(userId)
    }

}
```

“WebMvc.fn”, the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import org.springframework.web.servlet.function.RequestPredicate;
import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.ServerResponse;

import static org.springframework.web.servlet.function.RequestPredicates.accept;
import static org.springframework.web.servlet.function.RouterFunctions.route;

@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {

    private static final RequestPredicate ACCEPT_JSON =
accept(MediaType.APPLICATION_JSON);

    @Bean
    public RouterFunction<ServerResponse> routerFunction(MyUserHandler userHandler) {
        return route()
                .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
                .GET("/{user}/customers", ACCEPT_JSON, userHandler::getUserCustomers)
                .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
                .build();
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.MediaType
import org.springframework.web.servlet.function.RequestPredicates.accept
import org.springframework.web.servlet.function.RouterFunction
import org.springframework.web.servlet.function.RouterFunctions
import org.springframework.web.servlet.function.ServerResponse

@Configuration(proxyBeanMethods = false)
class MyRoutingConfiguration {

    @Bean
    fun routerFunction(userHandler: MyUserHandler): RouterFunction<ServerResponse> {
        return RouterFunctions.route()
            .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
            .GET("/{user}/customers", ACCEPT_JSON, userHandler::getUserCustomers)
            .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
            .build()
    }

    companion object {
        private val ACCEPT_JSON = accept(MediaType.APPLICATION_JSON)
    }

}
```

Java

```
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.function.ServerRequest;
import org.springframework.web.servlet.function.ServerResponse;

@Component
public class MyUserHandler {

    public ServerResponse getUser(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

    public ServerResponse getUserCustomers(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

    public ServerResponse deleteUser(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

}
```

Kotlin

```
import org.springframework.stereotype.Component
import org.springframework.web.servlet.function.ServerRequest
import org.springframework.web.servlet.function.ServerResponse

@Component
class MyUserHandler {

    fun getUser(request: ServerRequest?): ServerResponse {
        return ServerResponse.ok().build()
    }

    fun getUserCustomers(request: ServerRequest?): ServerResponse {
        return ServerResponse.ok().build()
    }

    fun deleteUser(request: ServerRequest?): ServerResponse {
        return ServerResponse.ok().build()
    }

}
```

Spring MVC is part of the core Spring Framework, and detailed information is available in the

reference documentation. There are also several guides that cover Spring MVC available at spring.io/guides.

TIP	You can define as many `RouterFunction` beans as you like to modularize the definition of the router. Beans can be ordered if you need to apply a precedence.

Spring MVC Auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications. It replaces the need for `@EnableWebMvc` and the two cannot be used together. In addition to Spring MVC's defaults, the auto-configuration provides the following features:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered later in this document).
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
- Support for `HttpMessageConverters` (covered later in this document).
- Automatic registration of `MessageCodesResolver` (covered later in this document).
- Static `index.html` support.
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered later in this document).

If you want to keep those Spring Boot MVC customizations and make more MVC customizations (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`.

If you want to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, and still keep the Spring Boot MVC customizations, you can declare a bean of type `WebMvcRegistrations` and use it to provide custom instances of those components. The custom instances will be subject to further initialization and configuration by Spring MVC. To participate in, and if desired, override that subsequent processing, a `WebMvcConfigurer` should be used.

If you do not want to use the auto-configuration and want to take complete control of Spring MVC, add your own `@Configuration` annotated with `@EnableWebMvc`. Alternatively, add your own `@Configuration`-annotated `DelegatingWebMvcConfiguration` as described in the Javadoc of `@EnableWebMvc`.

Spring MVC Conversion Service

Spring MVC uses a different `ConversionService` to the one used to convert values from your `application.properties` or `application.yaml` file. It means that `Period`, `Duration` and `DataSize` converters are not available and that `@DurationUnit` and `@DataSizeUnit` annotations will be ignored.

If you want to customize the `ConversionService` used by Spring MVC, you can provide a `WebMvcConfigurer` bean with an `addFormatters` method. From this method you can register any converter that you like, or you can delegate to the static methods available on `ApplicationConversionService`.

Conversion can also be customized using the `spring.mvc.format.*` configuration properties. When not configured, the following defaults are used:

Property	DateTimeFormatter
`spring.mvc.format.date`	`ofLocalizedDate(FormatStyle.SHORT)`
`spring.mvc.format.time`	`ofLocalizedTime(FormatStyle.SHORT)`
`spring.mvc.format.date-time`	`ofLocalizedDateTime(FormatStyle.SHORT)`

HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box. For example, objects can be automatically converted to JSON (by using the Jackson library) or XML (by using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in `UTF-8`.

If you need to add or customize converters, you can use Spring Boot's `HttpMessageConverters` class, as shown in the following listing:

Java

```
import org.springframework.boot.autoconfigure.http.HttpMessageConverters;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.converter.HttpMessageConverter;

@Configuration(proxyBeanMethods = false)
public class MyHttpMessageConvertersConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = new AdditionalHttpMessageConverter();
        HttpMessageConverter<?> another = new AnotherHttpMessageConverter();
        return new HttpMessageConverters(additional, another);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.http.HttpMessageConverters
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.converter.HttpMessageConverter

@Configuration(proxyBeanMethods = false)
class MyHttpMessageConvertersConfiguration {

    @Bean
    fun customConverters(): HttpMessageConverters {
        val additional: HttpMessageConverter<*> = AdditionalHttpMessageConverter()
        val another: HttpMessageConverter<*> = AnotherHttpMessageConverter()
        return HttpMessageConverters(additional, another)
    }

}
```

Any `HttpMessageConverter` bean that is present in the context is added to the list of converters. You can also override default converters in the same way.

MessageCodesResolver

Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: `MessageCodesResolver`. If you set the `spring.mvc.message-codes-resolver-format` property `PREFIX_ERROR_CODE` or `POSTFIX_ERROR_CODE`, Spring Boot creates one for you (see the enumeration in `DefaultMessageCodesResolver.Format`).

Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath or from the root of the `ServletContext`. It uses the `ResourceHttpRequestHandler` from Spring MVC so that you can modify that behavior by adding your own `WebMvcConfigurer` and overriding the `addResourceHandlers` method.

In a stand-alone web application, the default servlet from the container is not enabled. It can be enabled using the `server.servlet.register-default-servlet` property.

The default servlet acts as a fallback, serving content from the root of the `ServletContext` if Spring decides not to handle it. Most of the time, this does not happen (unless you modify the default MVC configuration), because Spring can always handle requests through the `DispatcherServlet`.

By default, resources are mapped on `/**`, but you can tune that with the `spring.mvc.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

Properties

```
spring.mvc.static-path-pattern=/resources/**
```

Yaml

```
spring:
  mvc:
    static-path-pattern: "/resources/**"
```

You can also customize the static resource locations by using the `spring.web.resources.static-locations` property (replacing the default values with a list of directory locations). The root servlet context path, `"/"`, is automatically added as a location as well.

In addition to the "standard" static resource locations mentioned earlier, a special case is made for Webjars content. By default, any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format. The path can be customized with the `spring.mvc.webjars-path-pattern` property.

TIP	Do not use the `src/main/webapp` directory if your application is packaged as a jar. Although this directory is a common standard, it works **only** with war packaging, and it is silently ignored by most build tools if you generate a jar.

Spring Boot also supports the advanced resource handling features provided by Spring MVC, allowing use cases such as cache-busting static resources or using version agnostic URLs for Webjars.

To use version agnostic URLs for Webjars, add the `webjars-locator-core` dependency. Then declare your Webjar. Using jQuery as an example, adding `"/webjars/jquery/jquery.min.js"` results in `"/webjars/jquery/x.y.z/jquery.min.js"` where `x.y.z` is the Webjar version.

NOTE	If you use JBoss, you need to declare the `webjars-locator-jboss-vfs` dependency instead of the `webjars-locator-core`. Otherwise, all Webjars resolve as a `404`.

To use cache busting, the following configuration configures a cache busting solution for all static resources, effectively adding a content hash, such as `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`, in URLs:

Properties

```
spring.web.resources.chain.strategy.content.enabled=true
spring.web.resources.chain.strategy.content.paths=/**
```

Yaml

```
spring:
  web:
    resources:
      chain:
        strategy:
          content:
            enabled: true
            paths: "/**"
```

NOTE Links to resources are rewritten in templates at runtime, thanks to a `ResourceUrlEncodingFilter` that is auto-configured for Thymeleaf and FreeMarker. You should manually declare this filter when using JSPs. Other template engines are currently not automatically supported but can be with custom template macros/helpers and the use of the `ResourceUrlProvider`.

When loading resources dynamically with, for example, a JavaScript module loader, renaming files is not an option. That is why other strategies are also supported and can be combined. A "fixed" strategy adds a static version string in the URL without changing the file name, as shown in the following example:

Properties

```
spring.web.resources.chain.strategy.content.enabled=true
spring.web.resources.chain.strategy.content.paths=/**
spring.web.resources.chain.strategy.fixed.enabled=true
spring.web.resources.chain.strategy.fixed.paths=/js/lib/
spring.web.resources.chain.strategy.fixed.version=v12
```

Yaml

```
spring:
  web:
    resources:
      chain:
        strategy:
          content:
            enabled: true
            paths: "/**"
          fixed:
            enabled: true
            paths: "/js/lib/"
            version: "v12"
```

With this configuration, JavaScript modules located under `"/js/lib/"` use a fixed versioning strategy (`"/v12/js/lib/mymodule.js"`), while other resources still use the content one (`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`).

See `WebProperties.Resources` for more supported options.

TIP	This feature has been thoroughly described in a dedicated blog post and in Spring Framework's reference documentation.

Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

Custom Favicon

As with other static resources, Spring Boot checks for a `favicon.ico` in the configured static content locations. If such a file is present, it is automatically used as the favicon of the application.

Path Matching and Content Negotiation

Spring MVC can map incoming HTTP requests to handlers by looking at the request path and matching it to the mappings defined in your application (for example, `@GetMapping` annotations on Controller methods).

Spring Boot chooses to disable suffix pattern matching by default, which means that requests like `"GET /projects/spring-boot.json"` will not be matched to `@GetMapping("/projects/spring-boot")` mappings. This is considered as a best practice for Spring MVC applications. This feature was mainly useful in the past for HTTP clients which did not send proper "Accept" request headers; we needed to make sure to send the correct Content Type to the client. Nowadays, Content Negotiation is much more reliable.

There are other ways to deal with HTTP clients that do not consistently send proper "Accept" request headers. Instead of using suffix matching, we can use a query parameter to ensure that requests like `"GET /projects/spring-boot?format=json"` will be mapped to `@GetMapping("/projects/spring-boot")`:

Properties

```
spring.mvc.contentnegotiation.favor-parameter=true
```

Yaml

```
spring:
  mvc:
    contentnegotiation:
      favor-parameter: true
```

Or if you prefer to use a different parameter name:

Properties

```
spring.mvc.contentnegotiation.favor-parameter=true
spring.mvc.contentnegotiation.parameter-name=myparam
```

Yaml

```
spring:
  mvc:
    contentnegotiation:
      favor-parameter: true
      parameter-name: "myparam"
```

Most standard media types are supported out-of-the-box, but you can also define new ones:

Properties

```
spring.mvc.contentnegotiation.media-types.markdown=text/markdown
```

Yaml

```
spring:
  mvc:
    contentnegotiation:
      media-types:
        markdown: "text/markdown"
```

As of Spring Framework 5.3, Spring MVC supports two strategies for matching request paths to controllers. By default, Spring Boot uses the `PathPatternParser` strategy. `PathPatternParser` is an optimized implementation but comes with some restrictions compared to the `AntPathMatcher` strategy. `PathPatternParser` restricts usage of some path pattern variants. It is also incompatible with configuring the `DispatcherServlet` with a path prefix (`spring.mvc.servlet.path`).

The strategy can be configured using the `spring.mvc.pathmatch.matching-strategy` configuration property, as shown in the following example:

Properties

```
spring.mvc.pathmatch.matching-strategy=ant-path-matcher
```

Yaml

```
spring:
  mvc:
    pathmatch:
      matching-strategy: "ant-path-matcher"
```

By default, Spring MVC will send a 404 Not Found error response if a handler is not found for a request. To have a `NoHandlerFoundException` thrown instead, set configprop:spring.mvc.throw-exception-if-no-handler-found to `true`. Note that, by default, the serving of static content is mapped to `/**` and will, therefore, provide a handler for all requests. For a `NoHandlerFoundException` to be thrown, you must also set `spring.mvc.static-path-pattern` to a more specific value such as `/resources/**` or set `spring.web.resources.add-mappings` to `false` to disable serving of static content entirely.

ConfigurableWebBindingInitializer

Spring MVC uses a `WebBindingInitializer` to initialize a `WebDataBinder` for a particular request. If you create your own `ConfigurableWebBindingInitializer @Bean`, Spring Boot automatically configures Spring MVC to use it.

Template Engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

TIP	If possible, JSPs should be avoided. There are several known limitations when using them with embedded servlet containers.

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

TIP	Depending on how you run your application, your IDE may order the classpath differently. Running your application in the IDE from its main method results in a different ordering than when you run your application by using Maven or Gradle or from its packaged jar. This can cause Spring Boot to fail to find the expected template. If you have this problem, you can reorder the classpath in the IDE to place the module's classes and resources first.

Error Handling

By default, Spring Boot provides an `/error` mapping that handles all errors in a sensible way, and it is registered as a "global" error page in the servlet container. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a "whitelabel" error view that renders the same data in HTML format (to customize it, add a `View` that resolves to `error`).

There are a number of `server.error` properties that can be set if you want to customize the default error handling behavior. See the "Server Properties" section of the Appendix.

To replace the default behavior completely, you can implement `ErrorController` and register a bean definition of that type or add a bean of type `ErrorAttributes` to use the existing mechanism but replace the contents.

TIP	The `BasicErrorController` can be used as a base class for a custom `ErrorController`. This is particularly useful if you want to add a handler for a new content type (the default is to handle `text/html` specifically and provide a fallback for everything else). To do so, extend `BasicErrorController`, add a public method with a `@RequestMapping` that has a `produces` attribute, and create a bean of your new type.

As of Spring Framework 6.0, RFC 7807 Problem Details is supported. Spring MVC can produce custom error messages with the `application/problem+json` media type, like:

```
{
  "type": "https://example.org/problems/unknown-project",
  "title": "Unknown project",
  "status": 404,
  "detail": "No project found for id 'spring-unknown'",
  "instance": "/projects/spring-unknown"
}
```

This support can be enabled by setting `spring.mvc.problemdetails.enabled` to `true`.

You can also define a class annotated with `@ControllerAdvice` to customize the JSON document to return for a particular controller and/or exception type, as shown in the following example:

Java

```
import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.http.HttpServletRequest;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

@ControllerAdvice(basePackageClasses = SomeController.class)
public class MyControllerAdvice extends ResponseEntityExceptionHandler {

    @ResponseBody
    @ExceptionHandler(MyException.class)
    public ResponseEntity<?> handleControllerException(HttpServletRequest request,
Throwable ex) {
        HttpStatus status = getStatus(request);
        return new ResponseEntity<>(new MyErrorBody(status.value(), ex.getMessage()),
status);
    }

    private HttpStatus getStatus(HttpServletRequest request) {
        Integer code = (Integer)
request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
        HttpStatus status = HttpStatus.resolve(code);
        return (status != null) ? status : HttpStatus.INTERNAL_SERVER_ERROR;
    }

}
```

Kotlin

```
import jakarta.servlet.RequestDispatcher
import jakarta.servlet.http.HttpServletRequest
import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.ControllerAdvice
import org.springframework.web.bind.annotation.ExceptionHandler
import org.springframework.web.bind.annotation.ResponseBody
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler

@ControllerAdvice(basePackageClasses = [SomeController::class])
class MyControllerAdvice : ResponseEntityExceptionHandler() {

    @ResponseBody
    @ExceptionHandler(MyException::class)
    fun handleControllerException(request: HttpServletRequest, ex: Throwable):
ResponseEntity<*> {
        val status = getStatus(request)
        return ResponseEntity(MyErrorBody(status.value(), ex.message), status)
    }

    private fun getStatus(request: HttpServletRequest): HttpStatus {
        val code = request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE) as Int
        val status = HttpStatus.resolve(code)
        return status ?: HttpStatus.INTERNAL_SERVER_ERROR
    }

}
```

In the preceding example, if `MyException` is thrown by a controller defined in the same package as `SomeController`, a JSON representation of the `MyErrorBody` POJO is used instead of the `ErrorAttributes` representation.

In some cases, errors handled at the controller level are not recorded by the metrics infrastructure. Applications can ensure that such exceptions are recorded with the request metrics by setting the handled exception as a request attribute:

Java

```
import jakarta.servlet.http.HttpServletRequest;

import org.springframework.boot.web.servlet.error.ErrorAttributes;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ExceptionHandler;

@Controller
public class MyController {

    @ExceptionHandler(CustomException.class)
    String handleCustomException(HttpServletRequest request, CustomException ex) {
        request.setAttribute(ErrorAttributes.ERROR_ATTRIBUTE, ex);
        return "errorView";
    }

}
```

Kotlin

```
import jakarta.servlet.http.HttpServletRequest
import org.springframework.boot.web.servlet.error.ErrorAttributes
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.ExceptionHandler

@Controller
class MyController {

    @ExceptionHandler(CustomException::class)
    fun handleCustomException(request: HttpServletRequest, ex: CustomException?):
String {
        request.setAttribute(ErrorAttributes.ERROR_ATTRIBUTE, ex)
        return "errorView"
    }

}
```

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` directory. Error pages can either be static HTML (that is, added under any of the static resource directories) or be built by using templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your directory structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

To map all `5xx` errors by using a FreeMarker template, your directory structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.ftlh
             +- <other templates>
```

For more complex mappings, you can also add beans that implement the `ErrorViewResolver` interface, as shown in the following example:

Java

```
import java.util.Map;

import jakarta.servlet.http.HttpServletRequest;

import org.springframework.boot.autoconfigure.web.servlet.error.ErrorViewResolver;
import org.springframework.http.HttpStatus;
import org.springframework.web.servlet.ModelAndView;

public class MyErrorViewResolver implements ErrorViewResolver {

    @Override
    public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus
status, Map<String, Object> model) {
        // Use the request or status to optionally return a ModelAndView
        if (status == HttpStatus.INSUFFICIENT_STORAGE) {
            // We could add custom model values here
            new ModelAndView("myview");
        }
        return null;
    }

}
```

Kotlin

```
import jakarta.servlet.http.HttpServletRequest
import org.springframework.boot.autoconfigure.web.servlet.error.ErrorViewResolver
import org.springframework.http.HttpStatus
import org.springframework.web.servlet.ModelAndView

class MyErrorViewResolver : ErrorViewResolver {

    override fun resolveErrorView(request: HttpServletRequest, status: HttpStatus,
            model: Map<String, Any>): ModelAndView? {
        // Use the request or status to optionally return a ModelAndView
        if (status == HttpStatus.INSUFFICIENT_STORAGE) {
            // We could add custom model values here
            return ModelAndView("myview")
        }
        return null
    }

}
```

You can also use regular Spring MVC features such as `@ExceptionHandler` methods and `@ControllerAdvice`. The `ErrorController` then picks up any unhandled exceptions.

Mapping Error Pages Outside of Spring MVC

For applications that do not use Spring MVC, you can use the `ErrorPageRegistrar` interface to directly register `ErrorPages`. This abstraction works directly with the underlying embedded servlet container and works even if you do not have a Spring MVC `DispatcherServlet`.

Java

```
import org.springframework.boot.web.server.ErrorPage;
import org.springframework.boot.web.server.ErrorPageRegistrar;
import org.springframework.boot.web.server.ErrorPageRegistry;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpStatus;

@Configuration(proxyBeanMethods = false)
public class MyErrorPagesConfiguration {

    @Bean
    public ErrorPageRegistrar errorPageRegistrar() {
        return this::registerErrorPages;
    }

    private void registerErrorPages(ErrorPageRegistry registry) {
        registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    }

}
```

Kotlin

```
import org.springframework.boot.web.server.ErrorPage
import org.springframework.boot.web.server.ErrorPageRegistrar
import org.springframework.boot.web.server.ErrorPageRegistry
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.HttpStatus

@Configuration(proxyBeanMethods = false)
class MyErrorPagesConfiguration {

    @Bean
    fun errorPageRegistrar(): ErrorPageRegistrar {
        return ErrorPageRegistrar { registry: ErrorPageRegistry ->
registerErrorPages(registry) }
    }

    private fun registerErrorPages(registry: ErrorPageRegistry) {
        registry.addErrorPages(ErrorPage(HttpStatus.BAD_REQUEST, "/400"))
    }

}
```

NOTE

If you register an `ErrorPage` with a path that ends up being handled by a `Filter` (as is common with some non-Spring web frameworks, like Jersey and Wicket), then the `Filter` has to be explicitly registered as an `ERROR` dispatcher, as shown in the following example:

Java

```
import java.util.EnumSet;

import jakarta.servlet.DispatcherType;

import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyFilterConfiguration {

    @Bean
    public FilterRegistrationBean<MyFilter> myFilter() {
        FilterRegistrationBean<MyFilter> registration = new
FilterRegistrationBean<>(new MyFilter());
        // ...
        registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.class));
        return registration;
    }

}
```

Kotlin

```
import jakarta.servlet.DispatcherType
import org.springframework.boot.web.servlet.FilterRegistrationBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import java.util.EnumSet

@Configuration(proxyBeanMethods = false)
class MyFilterConfiguration {

    @Bean
    fun myFilter(): FilterRegistrationBean<MyFilter> {
        val registration = FilterRegistrationBean(MyFilter())
        // ...
        registration.setDispatcherTypes(EnumSet.allOf(DispatcherType::class.java))
        return registration
    }

}
```

Note that the default `FilterRegistrationBean` does not include the `ERROR` dispatcher type.

Error Handling in a WAR Deployment

When deployed to a servlet container, Spring Boot uses its error page filter to forward a request

with an error status to the appropriate error page. This is necessary as the servlet specification does not provide an API for registering error pages. Depending on the container that you are deploying your war file to and the technologies that your application uses, some additional configuration may be required.

The error page filter can only forward the request to the correct error page if the response has not already been committed. By default, WebSphere Application Server 8.0 and later commits the response upon successful completion of a servlet's service method. You should disable this behavior by setting `com.ibm.ws.webcontainer.invokeFlushAfterService` to `false`.

CORS Support

Cross-origin resource sharing (CORS) is a W3C specification implemented by most browsers that lets you specify in a flexible way what kind of cross-domain requests are authorized, instead of using some less secure and less powerful approaches such as IFRAME or JSONP.

As of version 4.2, Spring MVC supports CORS. Using controller method CORS configuration with `@CrossOrigin` annotations in your Spring Boot application does not require any specific configuration. Global CORS configuration can be defined by registering a `WebMvcConfigurer` bean with a customized `addCorsMappings(CorsRegistry)` method, as shown in the following example:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration(proxyBeanMethods = false)
public class MyCorsConfiguration {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {

            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**");
            }

        };
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.web.servlet.config.annotation.CorsRegistry
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Configuration(proxyBeanMethods = false)
class MyCorsConfiguration {

    @Bean
    fun corsConfigurer(): WebMvcConfigurer {
        return object : WebMvcConfigurer {
            override fun addCorsMappings(registry: CorsRegistry) {
                registry.addMapping("/api/**")
            }
        }
    }

}
```

8.1.2. JAX-RS and Jersey

If you prefer the JAX-RS programming model for REST endpoints, you can use one of the available implementations instead of Spring MVC. Jersey and Apache CXF work quite well out of the box. CXF requires you to register its `Servlet` or `Filter` as a `@Bean` in your application context. Jersey has some native Spring support, so we also provide auto-configuration support for it in Spring Boot, together with a starter.

To get started with Jersey, include the `spring-boot-starter-jersey` as a dependency and then you need one `@Bean` of type `ResourceConfig` in which you register all the endpoints, as shown in the following example:

```
import org.glassfish.jersey.server.ResourceConfig;

import org.springframework.stereotype.Component;

@Component
public class MyJerseyConfig extends ResourceConfig {

    public MyJerseyConfig() {
        register(MyEndpoint.class);
    }

}
```

WARNING	Jersey's support for scanning executable archives is rather limited. For example, it cannot scan for endpoints in a package found in a fully executable jar file or in `WEB-INF/classes` when running an executable war file. To avoid this limitation, the `packages` method should not be used, and endpoints should be registered individually by using the `register` method, as shown in the preceding example.

For more advanced customizations, you can also register an arbitrary number of beans that implement `ResourceConfigCustomizer`.

All the registered endpoints should be `@Components` with HTTP resource annotations (`@GET` and others), as shown in the following example:

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

import org.springframework.stereotype.Component;

@Component
@Path("/hello")
public class MyEndpoint {

    @GET
    public String message() {
        return "Hello";
    }

}
```

Since the `Endpoint` is a Spring `@Component`, its lifecycle is managed by Spring and you can use the `@Autowired` annotation to inject dependencies and use the `@Value` annotation to inject external configuration. By default, the Jersey servlet is registered and mapped to `/*`. You can change the mapping by adding `@ApplicationPath` to your `ResourceConfig`.

By default, Jersey is set up as a servlet in a `@Bean` of type `ServletRegistrationBean` named `jerseyServletRegistration`. By default, the servlet is initialized lazily, but you can customize that behavior by setting `spring.jersey.servlet.load-on-startup`. You can disable or override that bean by creating one of your own with the same name. You can also use a filter instead of a servlet by setting `spring.jersey.type=filter` (in which case, the `@Bean` to replace or override is `jerseyFilterRegistration`). The filter has an `@Order`, which you can set with `spring.jersey.filter.order`. When using Jersey as a filter, a servlet that will handle any requests that are not intercepted by Jersey must be present. If your application does not contain such a servlet, you may want to enable the default servlet by setting `server.servlet.register-default-servlet` to `true`. Both the servlet and the filter registrations can be given init parameters by using `spring.jersey.init.*` to specify a map of properties.

8.1.3. Embedded Servlet Container Support

For servlet application, Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. Most developers use the appropriate "Starter" to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port `8080`.

Servlets, Filters, and Listeners

When using an embedded servlet container, you can register servlets, filters, and all the listeners (such as `HttpSessionListener`) from the servlet spec, either by using Spring beans or by scanning for servlet components.

Registering Servlets, Filters, and Listeners as Spring Beans

Any `Servlet`, `Filter`, or servlet `*Listener` instance that is a Spring bean is registered with the embedded container. This can be particularly convenient if you want to refer to a value from your `application.properties` during configuration.

By default, if the context contains only a single Servlet, it is mapped to `/`. In the case of multiple servlet beans, the bean name is used as a path prefix. Filters map to `/*`.

If convention-based mapping is not flexible enough, you can use the `ServletRegistrationBean`, `FilterRegistrationBean`, and `ServletListenerRegistrationBean` classes for complete control.

It is usually safe to leave filter beans unordered. If a specific order is required, you should annotate the `Filter` with `@Order` or make it implement `Ordered`. You cannot configure the order of a `Filter` by annotating its bean method with `@Order`. If you cannot change the `Filter` class to add `@Order` or implement `Ordered`, you must define a `FilterRegistrationBean` for the `Filter` and set the registration bean's order using the `setOrder(int)` method. Avoid configuring a filter that reads the request body at `Ordered.HIGHEST_PRECEDENCE`, since it might go against the character encoding configuration of your application. If a servlet filter wraps the request, it should be configured with an order that is less than or equal to `OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER`.

TIP	To see the order of every `Filter` in your application, enable debug level logging for the `web` logging group (`logging.level.web=debug`). Details of the registered filters, including their order and URL patterns, will then be logged at startup.

WARNING	Take care when registering `Filter` beans since they are initialized very early in the application lifecycle. If you need to register a `Filter` that interacts with other beans, consider using a `DelegatingFilterProxyRegistrationBean` instead.

Servlet Context Initialization

Embedded servlet containers do not directly execute the `jakarta.servlet.ServletContainerInitializer` interface or Spring's `org.springframework.web.WebApplicationInitializer` interface. This is an intentional design decision intended to reduce the risk that third party libraries designed to run inside a war may break Spring Boot applications.

If you need to perform servlet context initialization in a Spring Boot application, you should register a bean that implements the `org.springframework.boot.web.servlet.ServletContextInitializer` interface. The single `onStartup` method provides access to the `ServletContext` and, if necessary, can easily be used as an adapter to an existing `WebApplicationInitializer`.

Scanning for Servlets, Filters, and listeners

When using an embedded container, automatic registration of classes annotated with `@WebServlet`, `@WebFilter`, and `@WebListener` can be enabled by using `@ServletComponentScan`.

TIP	`@ServletComponentScan` has no effect in a standalone container, where the container's built-in discovery mechanisms are used instead.

The ServletWebServerApplicationContext

Under the hood, Spring Boot uses a different type of `ApplicationContext` for embedded servlet container support. The `ServletWebServerApplicationContext` is a special type of `WebApplicationContext` that bootstraps itself by searching for a single `ServletWebServerFactory` bean. Usually a `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` has been auto-configured.

NOTE	You usually do not need to be aware of these implementation classes. Most applications are auto-configured, and the appropriate `ApplicationContext` and `ServletWebServerFactory` are created on your behalf.

In an embedded container setup, the `ServletContext` is set as part of server startup which happens during application context initialization. Because of this beans in the `ApplicationContext` cannot be reliably initialized with a `ServletContext`. One way to get around this is to inject `ApplicationContext` as a dependency of the bean and access the `ServletContext` only when it is needed. Another way is to use a callback once the server has started. This can be done using an `ApplicationListener` which listens for the `ApplicationStartedEvent` as follows:

```
import jakarta.servlet.ServletContext;

import org.springframework.boot.context.event.ApplicationStartedEvent;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationListener;
import org.springframework.web.context.WebApplicationContext;

public class MyDemoBean implements ApplicationListener<ApplicationStartedEvent> {

    private ServletContext servletContext;

    @Override
    public void onApplicationEvent(ApplicationStartedEvent event) {
        ApplicationContext applicationContext = event.getApplicationContext();
        this.servletContext = ((WebApplicationContext)
applicationContext).getServletContext();
    }

}
```

Customizing Embedded Servlet Containers

Common servlet container settings can be configured by using Spring `Environment` properties. Usually, you would define the properties in your `application.properties` or `application.yaml` file.

Common server settings include:

- Network settings: Listen port for incoming HTTP requests (`server.port`), interface address to bind to `server.address`, and so on.
- Session settings: Whether the session is persistent (`server.servlet.session.persistent`), session timeout (`server.servlet.session.timeout`), location of session data (`server.servlet.session.store-dir`), and session-cookie configuration (`server.servlet.session.cookie.*`).
- Error management: Location of the error page (`server.error.path`) and so on.
- SSL
- HTTP compression

Spring Boot tries as much as possible to expose common settings, but this is not always possible. For those cases, dedicated namespaces offer server-specific customizations (see `server.tomcat` and `server.undertow`). For instance, access logs can be configured with specific features of the embedded servlet container.

TIP See the `ServerProperties` class for a complete list.

SameSite Cookies

The `SameSite` cookie attribute can be used by web browsers to control if and how cookies are

submitted in cross-site requests. The attribute is particularly relevant for modern web browsers which have started to change the default value that is used when the attribute is missing.

If you want to change the `SameSite` attribute of your session cookie, you can use the `server.servlet.session.cookie.same-site` property. This property is supported by auto-configured Tomcat, Jetty and Undertow servers. It is also used to configure Spring Session servlet based `SessionRepository` beans.

For example, if you want your session cookie to have a `SameSite` attribute of `None`, you can add the following to your `application.properties` or `application.yaml` file:

Properties

```
server.servlet.session.cookie.same-site=none
```

Yaml

```
server:
  servlet:
    session:
      cookie:
        same-site: "none"
```

If you want to change the `SameSite` attribute on other cookies added to your `HttpServletResponse`, you can use a `CookieSameSiteSupplier`. The `CookieSameSiteSupplier` is passed a `Cookie` and may return a `SameSite` value, or `null`.

There are a number of convenience factory and filter methods that you can use to quickly match specific cookies. For example, adding the following bean will automatically apply a `SameSite` of `Lax` for all cookies with a name that matches the regular expression `myapp.*`.

Java

```
import org.springframework.boot.web.servlet.server.CookieSameSiteSupplier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MySameSiteConfiguration {

    @Bean
    public CookieSameSiteSupplier applicationCookieSameSiteSupplier() {
        return CookieSameSiteSupplier.ofLax().whenHasNameMatching("myapp.*");
    }

}
```

Kotlin

```
import org.springframework.boot.web.servlet.server.CookieSameSiteSupplier
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MySameSiteConfiguration {

    @Bean
    fun applicationCookieSameSiteSupplier(): CookieSameSiteSupplier {
        return CookieSameSiteSupplier.ofLax().whenHasNameMatching("myapp.*")
    }

}
```

Character Encoding

The character encoding behavior of the embedded servlet container for request and response handling can be configured using the `server.servlet.encoding.*` configuration properties.

When a request's `Accept-Language` header indicates a locale for the request it will be automatically mapped to a charset by the servlet container. Each containers providers default locale to charset mappings and you should verify that they meet your application's needs. When they do not, use the `server.servlet.encoding.mapping` configuration property to customize the mappings, as shown in the following example:

Properties

```
server.servlet.encoding.mapping.ko=UTF-8
```

Yaml

```
server:
  servlet:
    encoding:
      mapping:
        ko: "UTF-8"
```

In the preceding example, the `ko` (Korean) locale has been mapped to `UTF-8`. This is equivalent to a `<locale-encoding-mapping-list>` entry in a `web.xml` file of a traditional war deployment.

Programmatic Customization

If you need to programmatically configure your embedded servlet container, you can register a Spring bean that implements the `WebServerFactoryCustomizer` interface. `WebServerFactoryCustomizer` provides access to the `ConfigurableServletWebServerFactory`, which includes numerous customization setter methods. The following example shows programmatically setting the port:

Java

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import
org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.stereotype.Component;

@Component
public class MyWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory server) {
        server.setPort(9000);
    }

}
```

Kotlin

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory
import org.springframework.stereotype.Component

@Component
class MyWebServerFactoryCustomizer :
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    override fun customize(server: ConfigurableServletWebServerFactory) {
        server.setPort(9000)
    }

}
```

`TomcatServletWebServerFactory`, `JettyServletWebServerFactory` and `UndertowServletWebServerFactory` are dedicated variants of `ConfigurableServletWebServerFactory` that have additional customization setter methods for Tomcat, Jetty and Undertow respectively. The following example shows how to customize `TomcatServletWebServerFactory` that provides access to Tomcat-specific configuration options:

Java

```
import java.time.Duration;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyTomcatWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    @Override
    public void customize(TomcatServletWebServerFactory server) {
        server.addConnectorCustomizers((connector) ->
connector.setAsyncTimeout(Duration.ofSeconds(20).toMillis()));
    }

}
```

Kotlin

```
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.stereotype.Component
import java.time.Duration

@Component
class MyTomcatWebServerFactoryCustomizer :
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    override fun customize(server: TomcatServletWebServerFactory) {
        server.addConnectorCustomizers({ connector -> connector.asyncTimeout =
Duration.ofSeconds(20).toMillis() })
    }

}
```

Customizing ConfigurableServletWebServerFactory Directly

For more advanced use cases that require you to extend from `ServletWebServerFactory`, you can expose a bean of such type yourself.

Setters are provided for many configuration options. Several protected method "hooks" are also provided should you need to do something more exotic. See the source code documentation for details.

NOTE	Auto-configured customizers are still applied on your custom factory, so use that option carefully.

JSP Limitations

When running a Spring Boot application that uses an embedded servlet container (and is packaged as an executable archive), there are some limitations in the JSP support.

- With Jetty and Tomcat, it should work if you use war packaging. An executable war will work when launched with `java -jar`, and will also be deployable to any standard container. JSPs are not supported when using an executable jar.
- Undertow does not support JSPs.
- Creating a custom `error.jsp` page does not override the default view for error handling. Custom error pages should be used instead.

8.2. Reactive Web Applications

Spring Boot simplifies development of reactive web applications by providing auto-configuration for Spring Webflux.

8.2.1. The "Spring WebFlux Framework"

Spring WebFlux is the new reactive web framework introduced in Spring Framework 5.0. Unlike Spring MVC, it does not require the servlet API, is fully asynchronous and non-blocking, and implements the Reactive Streams specification through the Reactor project.

Spring WebFlux comes in two flavors: functional and annotation-based. The annotation-based one is quite close to the Spring MVC model, as shown in the following example:

Java

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class MyRestController {

    private final UserRepository userRepository;

    private final CustomerRepository customerRepository;

    public MyRestController(UserRepository userRepository, CustomerRepository
customerRepository) {
        this.userRepository = userRepository;
        this.customerRepository = customerRepository;
    }

    @GetMapping("/{userId}")
    public Mono<User> getUser(@PathVariable Long userId) {
        return this.userRepository.findById(userId);
    }

    @GetMapping("/{userId}/customers")
    public Flux<Customer> getUserCustomers(@PathVariable Long userId) {
        return
this.userRepository.findById(userId).flatMapMany(this.customerRepository::findByUser);
    }

    @DeleteMapping("/{userId}")
    public Mono<Void> deleteUser(@PathVariable Long userId) {
        return this.userRepository.deleteById(userId);
    }

}
```

Kotlin

```
import org.springframework.web.bind.annotation.DeleteMapping
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import reactor.core.publisher.Flux
import reactor.core.publisher.Mono

@RestController
@RequestMapping("/users")
class MyRestController(private val userRepository: UserRepository, private val
customerRepository: CustomerRepository) {

    @GetMapping("/{userId}")
    fun getUser(@PathVariable userId: Long): Mono<User?> {
        return userRepository.findById(userId)
    }

    @GetMapping("/{userId}/customers")
    fun getUserCustomers(@PathVariable userId: Long): Flux<Customer> {
        return userRepository.findById(userId).flatMapMany { user: User? ->
            customerRepository.findByUser(user)
        }
    }

    @DeleteMapping("/{userId}")
    fun deleteUser(@PathVariable userId: Long): Mono<Void> {
        return userRepository.deleteById(userId)
    }

}
```

WebFlux is part of the Spring Framework and detailed information is available in its reference documentation.

“WebFlux.fn”, the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import org.springframework.web.reactive.function.server.RequestPredicate;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;

import static
org.springframework.web.reactive.function.server.RequestPredicates.accept;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;

@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {

    private static final RequestPredicate ACCEPT_JSON =
accept(MediaType.APPLICATION_JSON);

    @Bean
    public RouterFunction<ServerResponse> monoRouterFunction(MyUserHandler
userHandler) {
        return route()
                .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
                .GET("/{user}/customers", ACCEPT_JSON, userHandler::getUserCustomers)
                .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
                .build();
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.MediaType
import org.springframework.web.reactive.function.server.RequestPredicates.DELETE
import org.springframework.web.reactive.function.server.RequestPredicates.GET
import org.springframework.web.reactive.function.server.RequestPredicates.accept
import org.springframework.web.reactive.function.server.RouterFunction
import org.springframework.web.reactive.function.server.RouterFunctions
import org.springframework.web.reactive.function.server.ServerResponse

@Configuration(proxyBeanMethods = false)
class MyRoutingConfiguration {

    @Bean
    fun monoRouterFunction(userHandler: MyUserHandler): RouterFunction<ServerResponse>
{
        return RouterFunctions.route(
            GET("/{user}").and(ACCEPT_JSON), userHandler::getUser).andRoute(
            GET("/{user}/customers").and(ACCEPT_JSON),
userHandler::getUserCustomers).andRoute(
            DELETE("/{user}").and(ACCEPT_JSON), userHandler::deleteUser)
    }

    companion object {
        private val ACCEPT_JSON = accept(MediaType.APPLICATION_JSON)
    }

}
```

Java

```
import reactor.core.publisher.Mono;

import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;

@Component
public class MyUserHandler {

    public Mono<ServerResponse> getUser(ServerRequest request) {
        ...
    }

    public Mono<ServerResponse> getUserCustomers(ServerRequest request) {
        ...
    }

    public Mono<ServerResponse> deleteUser(ServerRequest request) {
        ...
    }

}
```

Kotlin

```
import org.springframework.stereotype.Component
import org.springframework.web.reactive.function.server.ServerRequest
import org.springframework.web.reactive.function.server.ServerResponse
import reactor.core.publisher.Mono

@Component
class MyUserHandler {

    fun getUser(request: ServerRequest?): Mono<ServerResponse> {
        return ServerResponse.ok().build()
    }

    fun getUserCustomers(request: ServerRequest?): Mono<ServerResponse> {
        return ServerResponse.ok().build()
    }

    fun deleteUser(request: ServerRequest?): Mono<ServerResponse> {
        return ServerResponse.ok().build()
    }

}
```

“WebFlux.fn” is part of the Spring Framework and detailed information is available in its reference

documentation.

TIP	You can define as many `RouterFunction` beans as you like to modularize the definition of the router. Beans can be ordered if you need to apply a precedence.

To get started, add the `spring-boot-starter-webflux` module to your application.

NOTE	Adding both `spring-boot-starter-web` and `spring-boot-starter-webflux` modules in your application results in Spring Boot auto-configuring Spring MVC, not WebFlux. This behavior has been chosen because many Spring developers add `spring-boot-starter-webflux` to their Spring MVC application to use the reactive `WebClient`. You can still enforce your choice by setting the chosen application type to `SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`.

Spring WebFlux Auto-configuration

Spring Boot provides auto-configuration for Spring WebFlux that works well with most applications.

The auto-configuration adds the following features on top of Spring's defaults:

- Configuring codecs for `HttpMessageReader` and `HttpMessageWriter` instances (described later in this document).
- Support for serving static resources, including support for WebJars (described later in this document).

If you want to keep Spring Boot WebFlux features and you want to add additional WebFlux configuration, you can add your own `@Configuration` class of type `WebFluxConfigurer` but **without** `@EnableWebFlux`.

If you want to take complete control of Spring WebFlux, you can add your own `@Configuration` annotated with `@EnableWebFlux`.

Spring WebFlux Conversion Service

If you want to customize the `ConversionService` used by Spring WebFlux, you can provide a `WebFluxConfigurer` bean with an `addFormatters` method.

Conversion can also be customized using the `spring.webflux.format.*` configuration properties. When not configured, the following defaults are used:

Property	`DateTimeFormatter`
`spring.webflux.format.date`	`ofLocalizedDate(FormatStyle.SHORT)`
`spring.webflux.format.time`	`ofLocalizedTime(FormatStyle.SHORT)`
`spring.webflux.format.date-time`	`ofLocalizedDateTime(FormatStyle.SHORT)`

HTTP Codecs with HttpMessageReaders and HttpMessageWriters

Spring WebFlux uses the `HttpMessageReader` and `HttpMessageWriter` interfaces to convert HTTP requests and responses. They are configured with `CodecConfigurer` to have sensible defaults by looking at the libraries available in your classpath.

Spring Boot provides dedicated configuration properties for codecs, `spring.codec.*`. It also applies further customization by using `CodecCustomizer` instances. For example, `spring.jackson.*` configuration keys are applied to the Jackson codec.

If you need to add or customize codecs, you can create a custom `CodecCustomizer` component, as shown in the following example:

Java

```
import org.springframework.boot.web.codec.CodecCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.codec.ServerSentEventHttpMessageReader;

@Configuration(proxyBeanMethods = false)
public class MyCodecsConfiguration {

    @Bean
    public CodecCustomizer myCodecCustomizer() {
        return (configurer) -> {
            configurer.registerDefaults(false);
            configurer.customCodecs().register(new
ServerSentEventHttpMessageReader());
            // ...
        };
    }

}
```

Kotlin

```
import org.springframework.boot.web.codec.CodecCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.http.codec.CodecConfigurer
import org.springframework.http.codec.ServerSentEventHttpMessageReader

class MyCodecsConfiguration {

    @Bean
    fun myCodecCustomizer(): CodecCustomizer {
        return CodecCustomizer { configurer: CodecConfigurer ->
            configurer.registerDefaults(false)
            configurer.customCodecs().register(ServerSentEventHttpMessageReader())
        }
    }

}
```

You can also leverage Boot's custom JSON serializers and deserializers.

Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath. It uses the `ResourceWebHandler` from Spring WebFlux so that you can modify that behavior by adding your own `WebFluxConfigurer` and overriding the `addResourceHandlers` method.

By default, resources are mapped on `/**`, but you can tune that by setting the `spring.webflux.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

Properties

```
spring.webflux.static-path-pattern=/resources/**
```

Yaml

```
spring:
  webflux:
    static-path-pattern: "/resources/**"
```

You can also customize the static resource locations by using `spring.web.resources.static-locations`. Doing so replaces the default values with a list of directory locations. If you do so, the default welcome page detection switches to your custom locations. So, if there is an `index.html` in any of your locations on startup, it is the home page of the application.

In addition to the "standard" static resource locations listed earlier, a special case is made for Webjars content. By default, any resources with a path in `/webjars/**` are served from jar files if

they are packaged in the Webjars format. The path can be customized with the `spring.webflux.webjars-path-pattern` property.

TIP	Spring WebFlux applications do not strictly depend on the servlet API, so they cannot be deployed as war files and do not use the `src/main/webapp` directory.

Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

Template Engines

As well as REST web services, you can also use Spring WebFlux to serve dynamic HTML content. Spring WebFlux supports a variety of templating technologies, including Thymeleaf, FreeMarker, and Mustache.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Thymeleaf
- Mustache

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

Error Handling

Spring Boot provides a `WebExceptionHandler` that handles all errors in a sensible way. Its position in the processing order is immediately before the handlers provided by WebFlux, which are considered last. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a "whitelabel" error handler that renders the same data in HTML format. You can also provide your own HTML templates to display errors (see the next section).

Before customizing error handling in Spring Boot directly, you can leverage the RFC 7807 Problem Details support in Spring WebFlux. Spring WebFlux can produce custom error messages with the `application/problem+json` media type, like:

```
{
  "type": "https://example.org/problems/unknown-project",
  "title": "Unknown project",
  "status": 404,
  "detail": "No project found for id 'spring-unknown'",
  "instance": "/projects/spring-unknown"
}
```

This support can be enabled by setting `spring.webflux.problemdetails.enabled` to `true`.

The first step to customizing this feature often involves using the existing mechanism but replacing or augmenting the error contents. For that, you can add a bean of type `ErrorAttributes`.

To change the error handling behavior, you can implement `ErrorWebExceptionHandler` and register a bean definition of that type. Because an `ErrorWebExceptionHandler` is quite low-level, Spring Boot also provides a convenient `AbstractErrorWebExceptionHandler` to let you handle errors in a WebFlux functional way, as shown in the following example:

Java

```
import reactor.core.publisher.Mono;

import org.springframework.boot.autoconfigure.web.WebProperties;
import
org.springframework.boot.autoconfigure.web.reactive.error.AbstractErrorWebExceptionHan
dler;
import org.springframework.boot.web.reactive.error.ErrorAttributes;
import org.springframework.context.ApplicationContext;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.codec.ServerCodecConfigurer;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import org.springframework.web.reactive.function.server.ServerResponse.BodyBuilder;

@Component
public class MyErrorWebExceptionHandler extends AbstractErrorWebExceptionHandler {

    public MyErrorWebExceptionHandler(ErrorAttributes errorAttributes, WebProperties
webProperties,
            ApplicationContext applicationContext, ServerCodecConfigurer
serverCodecConfigurer) {
        super(errorAttributes, webProperties.getResources(), applicationContext);
        setMessageReaders(serverCodecConfigurer.getReaders());
        setMessageWriters(serverCodecConfigurer.getWriters());
    }

    @Override
    protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes
errorAttributes) {
        return RouterFunctions.route(this::acceptsXml, this::handleErrorAsXml);
    }

    private boolean acceptsXml(ServerRequest request) {
        return request.headers().accept().contains(MediaType.APPLICATION_XML);
    }

    public Mono<ServerResponse> handleErrorAsXml(ServerRequest request) {
        BodyBuilder builder = ServerResponse.status(HttpStatus.INTERNAL_SERVER_ERROR);
        // ... additional builder calls
        return builder.build();
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.web.WebProperties
import
org.springframework.boot.autoconfigure.web.reactive.error.AbstractErrorWebExceptionHan
dler
import org.springframework.boot.web.reactive.error.ErrorAttributes
import org.springframework.context.ApplicationContext
import org.springframework.http.HttpStatus
import org.springframework.http.MediaType
import org.springframework.http.codec.ServerCodecConfigurer
import org.springframework.stereotype.Component
import org.springframework.web.reactive.function.server.RouterFunction
import org.springframework.web.reactive.function.server.RouterFunctions
import org.springframework.web.reactive.function.server.ServerRequest
import org.springframework.web.reactive.function.server.ServerResponse
import reactor.core.publisher.Mono

@Component
class MyErrorWebExceptionHandler(
        errorAttributes: ErrorAttributes, webProperties: WebProperties,
        applicationContext: ApplicationContext, serverCodecConfigurer:
ServerCodecConfigurer
) : AbstractErrorWebExceptionHandler(errorAttributes, webProperties.resources,
applicationContext) {

    init {
        setMessageReaders(serverCodecConfigurer.readers)
        setMessageWriters(serverCodecConfigurer.writers)
    }

    override fun getRoutingFunction(errorAttributes: ErrorAttributes):
RouterFunction<ServerResponse> {
        return RouterFunctions.route(this::acceptsXml, this::handleErrorAsXml)
    }

    private fun acceptsXml(request: ServerRequest): Boolean {
        return request.headers().accept().contains(MediaType.APPLICATION_XML)
    }

    fun handleErrorAsXml(request: ServerRequest): Mono<ServerResponse> {
        val builder = ServerResponse.status(HttpStatus.INTERNAL_SERVER_ERROR)
        // ... additional builder calls
        return builder.build()
    }

}
```

For a more complete picture, you can also subclass `DefaultErrorWebExceptionHandler` directly and override specific methods.

In some cases, errors handled at the controller or handler function level are not recorded by the metrics infrastructure. Applications can ensure that such exceptions are recorded with the request metrics by setting the handled exception as a request attribute:

Java

```
import org.springframework.boot.web.reactive.error.ErrorAttributes;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.reactive.result.view.Rendering;
import org.springframework.web.server.ServerWebExchange;

@Controller
public class MyExceptionHandlingController {

    @GetMapping("/profile")
    public Rendering userProfile() {
        // ...
        throw new IllegalStateException();
    }

    @ExceptionHandler(IllegalStateException.class)
    public Rendering handleIllegalState(ServerWebExchange exchange,
IllegalStateException exc) {
        exchange.getAttributes().putIfAbsent(ErrorAttributes.ERROR_ATTRIBUTE, exc);
        return Rendering.view("errorView").modelAttribute("message",
exc.getMessage()).build();
    }

}
```

Kotlin

```
import org.springframework.boot.web.reactive.error.ErrorAttributes
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.ExceptionHandler
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.reactive.result.view.Rendering
import org.springframework.web.server.ServerWebExchange

@Controller
class MyExceptionHandlingController {

    @GetMapping("/profile")
    fun userProfile(): Rendering {
        // ...
        throw IllegalStateException()
    }

    @ExceptionHandler(IllegalStateException::class)
    fun handleIllegalState(exchange: ServerWebExchange, exc: IllegalStateException):
Rendering {
        exchange.attributes.putIfAbsent(ErrorAttributes.ERROR_ATTRIBUTE, exc)
        return Rendering.view("errorView").modelAttribute("message", exc.message ?:
"").build()
    }

}
```

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add views that resolve from `error/*`, for example by adding files to a `/error` directory. Error pages can either be static HTML (that is, added under any of the static resource directories) or built with templates. The name of the file should be the exact status code, a status code series mask, or `error` for a default if nothing else matches. Note that the path to the default error view is `error/error`, whereas with Spring MVC the default error view is `error`.

For example, to map `404` to a static HTML file, your directory structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

To map all `5xx` errors by using a Mustache template, your directory structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.mustache
             +- <other templates>
```

Web Filters

Spring WebFlux provides a `WebFilter` interface that can be implemented to filter HTTP request-response exchanges. `WebFilter` beans found in the application context will be automatically used to filter each exchange.

Where the order of the filters is important they can implement `Ordered` or be annotated with `@Order`. Spring Boot auto-configuration may configure web filters for you. When it does so, the orders shown in the following table will be used:

Web Filter	Order
`ServerHttpObservationFilter` (Micrometer Observability)	`Ordered.HIGHEST_PRECEDENCE + 1`
`WebFilterChainProxy` (Spring Security)	`-100`
`HttpExchangesWebFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

8.2.2. Embedded Reactive Server Support

Spring Boot includes support for the following embedded reactive web servers: Reactor Netty, Tomcat, Jetty, and Undertow. Most developers use the appropriate "Starter" to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port 8080.

Customizing Reactive Servers

Common reactive web server settings can be configured by using Spring `Environment` properties. Usually, you would define the properties in your `application.properties` or `application.yaml` file.

Common server settings include:

- Network settings: Listen port for incoming HTTP requests (`server.port`), interface address to bind to `server.address`, and so on.
- Error management: Location of the error page (`server.error.path`) and so on.
- SSL
- HTTP compression

Spring Boot tries as much as possible to expose common settings, but this is not always possible. For those cases, dedicated namespaces such as `server.netty.*` offer server-specific customizations.

TIP	See the `ServerProperties` class for a complete list.

Programmatic Customization

If you need to programmatically configure your reactive web server, you can register a Spring bean that implements the `WebServerFactoryCustomizer` interface. `WebServerFactoryCustomizer` provides access to the `ConfigurableReactiveWebServerFactory`, which includes numerous customization setter methods. The following example shows programmatically setting the port:

Java

```
import
org.springframework.boot.web.reactive.server.ConfigurableReactiveWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<ConfigurableReactiveWebServerFactory> {

    @Override
    public void customize(ConfigurableReactiveWebServerFactory server) {
        server.setPort(9000);
    }

}
```

Kotlin

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import
org.springframework.boot.web.reactive.server.ConfigurableReactiveWebServerFactory
import org.springframework.stereotype.Component

@Component
class MyWebServerFactoryCustomizer :
WebServerFactoryCustomizer<ConfigurableReactiveWebServerFactory> {

    override fun customize(server: ConfigurableReactiveWebServerFactory) {
        server.setPort(9000)
    }

}
```

`JettyReactiveWebServerFactory`, `NettyReactiveWebServerFactory`, `TomcatReactiveWebServerFactory`, and `UndertowServletWebServerFactory` are dedicated variants of `ConfigurableReactiveWebServerFactory`

that have additional customization setter methods for Jetty, Reactor Netty, Tomcat, and Undertow respectively. The following example shows how to customize `NettyReactiveWebServerFactory` that provides access to Reactor Netty-specific configuration options:

Java

```
import java.time.Duration;

import org.springframework.boot.web.embedded.netty.NettyReactiveWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyNettyWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<NettyReactiveWebServerFactory> {

    @Override
    public void customize(NettyReactiveWebServerFactory factory) {
        factory.addServerCustomizers((server) ->
server.idleTimeout(Duration.ofSeconds(20)));
    }

}
```

Kotlin

```
import org.springframework.boot.web.embedded.netty.NettyReactiveWebServerFactory
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.stereotype.Component
import java.time.Duration

@Component
class MyNettyWebServerFactoryCustomizer :
WebServerFactoryCustomizer<NettyReactiveWebServerFactory> {

    override fun customize(factory: NettyReactiveWebServerFactory) {
        factory.addServerCustomizers({ server ->
server.idleTimeout(Duration.ofSeconds(20)) })
    }

}
```

Customizing ConfigurableReactiveWebServerFactory Directly

For more advanced use cases that require you to extend from `ReactiveWebServerFactory`, you can expose a bean of such type yourself.

Setters are provided for many configuration options. Several protected method "hooks" are also provided should you need to do something more exotic. See the source code documentation for details.

NOTE	Auto-configured customizers are still applied on your custom factory, so use that option carefully.

8.2.3. Reactive Server Resources Configuration

When auto-configuring a Reactor Netty or Jetty server, Spring Boot will create specific beans that will provide HTTP resources to the server instance: `ReactorResourceFactory` or `JettyResourceFactory`.

By default, those resources will be also shared with the Reactor Netty and Jetty clients for optimal performances, given:

- the same technology is used for server and client
- the client instance is built using the `WebClient.Builder` bean auto-configured by Spring Boot

Developers can override the resource configuration for Jetty and Reactor Netty by providing a custom `ReactorResourceFactory` or `JettyResourceFactory` bean - this will be applied to both clients and servers.

You can learn more about the resource configuration on the client side in the WebClient Runtime section.

8.3. Graceful Shutdown

Graceful shutdown is supported with all four embedded web servers (Jetty, Reactor Netty, Tomcat, and Undertow) and with both reactive and servlet-based web applications. It occurs as part of closing the application context and is performed in the earliest phase of stopping `SmartLifecycle` beans. This stop processing uses a timeout which provides a grace period during which existing requests will be allowed to complete but no new requests will be permitted. The exact way in which new requests are not permitted varies depending on the web server that is being used. Jetty, Reactor Netty, and Tomcat will stop accepting requests at the network layer. Undertow will accept requests but respond immediately with a service unavailable (503) response.

NOTE	Graceful shutdown with Tomcat requires Tomcat 9.0.33 or later.

To enable graceful shutdown, configure the `server.shutdown` property, as shown in the following example:

Properties

```
server.shutdown=graceful
```

Yaml

```
server:
  shutdown: "graceful"
```

To configure the timeout period, configure the `spring.lifecycle.timeout-per-shutdown-phase`

property, as shown in the following example:

Properties

```
spring.lifecycle.timeout-per-shutdown-phase=20s
```

Yaml

```
spring:
  lifecycle:
    timeout-per-shutdown-phase: "20s"
```

IMPORTANT	Using graceful shutdown with your IDE may not work properly if it does not send a proper `SIGTERM` signal. See the documentation of your IDE for more details.

8.4. Spring Security

If Spring Security is on the classpath, then web applications are secured by default. Spring Boot relies on Spring Security's content-negotiation strategy to determine whether to use `httpBasic` or `formLogin`. To add method-level security to a web application, you can also add `@EnableGlobalMethodSecurity` with your desired settings. Additional information can be found in the Spring Security Reference Guide.

The default `UserDetailsService` has a single user. The user name is `user`, and the password is random and is printed at WARN level when the application starts, as shown in the following example:

```
Using generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35

This generated password is for development use only. Your security configuration must
be updated before running your application in production.
```

NOTE	If you fine-tune your logging configuration, ensure that the `org.springframework.boot.autoconfigure.security` category is set to log `WARN`-level messages. Otherwise, the default password is not printed.

You can change the username and password by providing a `spring.security.user.name` and `spring.security.user.password`.

The basic features you get by default in a web application are:

- A `UserDetailsService` (or `ReactiveUserDetailsService` in case of a WebFlux application) bean with in-memory store and a single user with a generated password (see `SecurityProperties.User` for the properties of the user).
- Form-based login or HTTP Basic security (depending on the `Accept` header in the request) for the

entire application (including actuator endpoints if actuator is on the classpath).

- A `DefaultAuthenticationEventPublisher` for publishing authentication events.

You can provide a different `AuthenticationEventPublisher` by adding a bean for it.

8.4.1. MVC Security

The default security configuration is implemented in `SecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `SecurityAutoConfiguration` imports `SpringBootWebSecurityConfiguration` for web security and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications. To switch off the default web application security configuration completely or to combine multiple Spring Security components such as OAuth2 Client and Resource Server, add a bean of type `SecurityFilterChain` (doing so does not disable the `UserDetailsService` configuration or Actuator's security).

To also switch off the `UserDetailsService` configuration, you can add a bean of type `UserDetailsService`, `AuthenticationProvider`, or `AuthenticationManager`.

Access rules can be overridden by adding a custom `SecurityFilterChain` bean. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `RequestMatcher` that is based on the `management.endpoints.web.base-path` property. `PathRequest` can be used to create a `RequestMatcher` for resources in commonly used locations.

8.4.2. WebFlux Security

Similar to Spring MVC applications, you can secure your WebFlux applications by adding the `spring-boot-starter-security` dependency. The default security configuration is implemented in `ReactiveSecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `ReactiveSecurityAutoConfiguration` imports `WebFluxSecurityConfiguration` for web security and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications. To switch off the default web application security configuration completely, you can add a bean of type `WebFilterChainProxy` (doing so does not disable the `UserDetailsService` configuration or Actuator's security).

To also switch off the `UserDetailsService` configuration, you can add a bean of type `ReactiveUserDetailsService` or `ReactiveAuthenticationManager`.

Access rules and the use of multiple Spring Security components such as OAuth 2 Client and Resource Server can be configured by adding a custom `SecurityWebFilterChain` bean. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `ServerWebExchangeMatcher` that is based on the `management.endpoints.web.base-path` property.

`PathRequest` can be used to create a `ServerWebExchangeMatcher` for resources in commonly used locations.

For example, you can customize your security configuration by adding something like:

Java

```
import org.springframework.boot.autoconfigure.security.reactive.PathRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration(proxyBeanMethods = false)
public class MyWebFluxSecurityConfiguration {

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http.authorizeExchange((exchange) -> {

exchange.matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll();
            exchange.pathMatchers("/foo", "/bar").authenticated();
        });
        http.formLogin(withDefaults());
        return http.build();
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.security.reactive.PathRequest
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.config.Customizer
import org.springframework.security.config.Customizer.withDefaults
import org.springframework.security.config.web.server.ServerHttpSecurity
import org.springframework.security.web.server.SecurityWebFilterChain

@Configuration(proxyBeanMethods = false)
class MyWebFluxSecurityConfiguration {

    @Bean
    fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
        http.authorizeExchange { spec ->

spec.matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
            spec.pathMatchers("/foo", "/bar").authenticated()
        }
        http.formLogin(withDefaults())
        return http.build()
    }

}
```

8.4.3. OAuth2

OAuth2 is a widely used authorization framework that is supported by Spring.

Client

If you have `spring-security-oauth2-client` on your classpath, you can take advantage of some auto-configuration to set up OAuth2/Open ID Connect clients. This configuration makes use of the properties under `OAuth2ClientProperties`. The same properties are applicable to both servlet and reactive applications.

You can register multiple OAuth2 clients and providers under the `spring.security.oauth2.client` prefix, as shown in the following example:

Properties

```
spring.security.oauth2.client.registration.my-login-client.client-id=abcd
spring.security.oauth2.client.registration.my-login-client.client-secret=password
spring.security.oauth2.client.registration.my-login-client.client-name=Client for
OpenID Connect
spring.security.oauth2.client.registration.my-login-client.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-login-
client.scope=openid,profile,email,phone,address
spring.security.oauth2.client.registration.my-login-client.redirect-
```

```
uri={baseUrl}/login/oauth2/code/{registrationId}
spring.security.oauth2.client.registration.my-login-client.client-authentication-
method=client_secret_basic
spring.security.oauth2.client.registration.my-login-client.authorization-grant-
type=authorization_code

spring.security.oauth2.client.registration.my-client-1.client-id=abcd
spring.security.oauth2.client.registration.my-client-1.client-secret=password
spring.security.oauth2.client.registration.my-client-1.client-name=Client for user
scope
spring.security.oauth2.client.registration.my-client-1.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-1.scope=user
spring.security.oauth2.client.registration.my-client-1.redirect-
uri={baseUrl}/authorized/user
spring.security.oauth2.client.registration.my-client-1.client-authentication-
method=client_secret_basic
spring.security.oauth2.client.registration.my-client-1.authorization-grant-
type=authorization_code

spring.security.oauth2.client.registration.my-client-2.client-id=abcd
spring.security.oauth2.client.registration.my-client-2.client-secret=password
spring.security.oauth2.client.registration.my-client-2.client-name=Client for email
scope
spring.security.oauth2.client.registration.my-client-2.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-2.scope=email
spring.security.oauth2.client.registration.my-client-2.redirect-
uri={baseUrl}/authorized/email
spring.security.oauth2.client.registration.my-client-2.client-authentication-
method=client_secret_basic
spring.security.oauth2.client.registration.my-client-2.authorization-grant-
type=authorization_code

spring.security.oauth2.client.provider.my-oauth-provider.authorization-uri=https://my-
auth-server.com/oauth2/authorize
spring.security.oauth2.client.provider.my-oauth-provider.token-uri=https://my-auth-
server.com/oauth2/token
spring.security.oauth2.client.provider.my-oauth-provider.user-info-uri=https://my-
auth-server.com/userinfo
spring.security.oauth2.client.provider.my-oauth-provider.user-info-authentication-
method=header
spring.security.oauth2.client.provider.my-oauth-provider.jwk-set-uri=https://my-auth-
server.com/oauth2/jwks
spring.security.oauth2.client.provider.my-oauth-provider.user-name-attribute=name
```

Yaml

```
spring:
  security:
    oauth2:
      client:
        registration:
          my-login-client:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for OpenID Connect"
            provider: "my-oauth-provider"
            scope: "openid,profile,email,phone,address"
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            client-authentication-method: "client_secret_basic"
            authorization-grant-type: "authorization_code"

          my-client-1:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for user scope"
            provider: "my-oauth-provider"
            scope: "user"
            redirect-uri: "{baseUrl}/authorized/user"
            client-authentication-method: "client_secret_basic"
            authorization-grant-type: "authorization_code"

          my-client-2:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for email scope"
            provider: "my-oauth-provider"
            scope: "email"
            redirect-uri: "{baseUrl}/authorized/email"
            client-authentication-method: "client_secret_basic"
            authorization-grant-type: "authorization_code"

        provider:
          my-oauth-provider:
            authorization-uri: "https://my-auth-server.com/oauth2/authorize"
            token-uri: "https://my-auth-server.com/oauth2/token"
            user-info-uri: "https://my-auth-server.com/userinfo"
            user-info-authentication-method: "header"
            jwk-set-uri: "https://my-auth-server.com/oauth2/jwks"
            user-name-attribute: "name"
```

For OpenID Connect providers that support OpenID Connect discovery, the configuration can be further simplified. The provider needs to be configured with an `issuer-uri` which is the URI that it asserts as its Issuer Identifier. For example, if the `issuer-uri` provided is "https://example.com", then an "OpenID Provider Configuration Request" will be made to "https://example.com/.well-

known/openid-configuration". The result is expected to be an "OpenID Provider Configuration Response". The following example shows how an OpenID Connect Provider can be configured with the `issuer-uri`:

Properties

```
spring.security.oauth2.client.provider.oidc-provider.issuer-uri=https://dev-
123456.oktapreview.com/oauth2/default/
```

Yaml

```
spring:
  security:
    oauth2:
      client:
        provider:
          oidc-provider:
            issuer-uri: "https://dev-123456.oktapreview.com/oauth2/default/"
```

By default, Spring Security's `OAuth2LoginAuthenticationFilter` only processes URLs matching `/login/oauth2/code/*`. If you want to customize the `redirect-uri` to use a different pattern, you need to provide configuration to process that custom pattern. For example, for servlet applications, you can add your own `SecurityFilterChain` that resembles the following:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration(proxyBeanMethods = false)
@EnableWebSecurity
public class MyOAuthClientConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http
            .authorizeHttpRequests((requests) -> requests
                .anyRequest().authenticated()
            )
            .oauth2Login((login) -> login
                .redirectionEndpoint((endpoint) -> endpoint
                    .baseUri("/login/oauth2/callback/*")
                )
            );
        return http.build();
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
import org.springframework.security.config.annotation.web.invoke
import org.springframework.security.web.SecurityFilterChain

@Configuration(proxyBeanMethods = false)
@EnableWebSecurity
open class MyOAuthClientConfiguration {

    @Bean
    open fun securityFilterChain(http: HttpSecurity): SecurityFilterChain {
        http {
            authorizeHttpRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2Login {
                redirectionEndpoint {
                    baseUri = "/login/oauth2/callback/*"
                }
            }
        }
        return http.build()
    }

}
```

TIP

Spring Boot auto-configures an `InMemoryOAuth2AuthorizedClientService` which is used by Spring Security for the management of client registrations. The `InMemoryOAuth2AuthorizedClientService` has limited capabilities and we recommend using it only for development environments. For production environments, consider using a `JdbcOAuth2AuthorizedClientService` or creating your own implementation of `OAuth2AuthorizedClientService`.

OAuth2 Client Registration for Common Providers

For common OAuth2 and OpenID providers, including Google, Github, Facebook, and Okta, we provide a set of provider defaults (`google`, `github`, `facebook`, and `okta`, respectively).

If you do not need to customize these providers, you can set the `provider` attribute to the one for which you need to infer defaults. Also, if the key for the client registration matches a default supported provider, Spring Boot infers that as well.

In other words, the two configurations in the following example use the Google provider:

Properties

```
spring.security.oauth2.client.registration.my-client.client-id=abcd
spring.security.oauth2.client.registration.my-client.client-secret=password
spring.security.oauth2.client.registration.my-client.provider=google
spring.security.oauth2.client.registration.google.client-id=abcd
spring.security.oauth2.client.registration.google.client-secret=password
```

Yaml

```
spring:
  security:
    oauth2:
      client:
        registration:
          my-client:
            client-id: "abcd"
            client-secret: "password"
            provider: "google"
          google:
            client-id: "abcd"
            client-secret: "password"
```

Resource Server

If you have `spring-security-oauth2-resource-server` on your classpath, Spring Boot can set up an OAuth2 Resource Server. For JWT configuration, a JWK Set URI or OIDC Issuer URI needs to be specified, as shown in the following examples:

Properties

```
spring.security.oauth2.resourceserver.jwt.jwk-set-
uri=https://example.com/oauth2/default/v1/keys
```

Yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: "https://example.com/oauth2/default/v1/keys"
```

Properties

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=https://dev-
123456.oktapreview.com/oauth2/default/
```

Yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: "https://dev-123456.oktapreview.com/oauth2/default/"
```

NOTE If the authorization server does not support a JWK Set URI, you can configure the resource server with the Public Key used for verifying the signature of the JWT. This can be done using the `spring.security.oauth2.resourceserver.jwt.public-key-location` property, where the value needs to point to a file containing the public key in the PEM-encoded x509 format.

The `spring.security.oauth2.resourceserver.jwt.audiences` property can be used to specify the expected values of the aud claim in JWTs. For example, to require JWTs to contain an aud claim with the value `my-audience`:

Properties

```
spring.security.oauth2.resourceserver.jwt.audiences[0]=my-audience
```

Yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          audiences:
            - "my-audience"
```

The same properties are applicable for both servlet and reactive applications. Alternatively, you can define your own `JwtDecoder` bean for servlet applications or a `ReactiveJwtDecoder` for reactive applications.

In cases where opaque tokens are used instead of JWTs, you can configure the following properties to validate tokens through introspection:

Properties

```
spring.security.oauth2.resourceserver.opaquetoken.introspection-
uri=https://example.com/check-token
spring.security.oauth2.resourceserver.opaquetoken.client-id=my-client-id
spring.security.oauth2.resourceserver.opaquetoken.client-secret=my-client-secret
```

Yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        opaquetoken:
          introspection-uri: "https://example.com/check-token"
          client-id: "my-client-id"
          client-secret: "my-client-secret"
```

Again, the same properties are applicable for both servlet and reactive applications. Alternatively, you can define your own `OpaqueTokenIntrospector` bean for servlet applications or a `ReactiveOpaqueTokenIntrospector` for reactive applications.

Authorization Server

If you have `spring-security-oauth2-authorization-server` on your classpath, you can take advantage of some auto-configuration to set up a Servlet-based OAuth2 Authorization Server.

You can register multiple OAuth2 clients under the `spring.security.oauth2.authorizationserver.client` prefix, as shown in the following example:

Properties

```
spring.security.oauth2.authorizationserver.client.my-client-1.registration.client-
id=abcd
spring.security.oauth2.authorizationserver.client.my-client-1.registration.client-
secret={noop}secret1
spring.security.oauth2.authorizationserver.client.my-client-1.registration.client-
authentication-methods[0]=client_secret_basic
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.authorization-grant-types[0]=authorization_code
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.authorization-grant-types[1]=refresh_token
spring.security.oauth2.authorizationserver.client.my-client-1.registration.redirect-
uris[0]=https://my-client-1.com/login/oauth2/code/abcd
spring.security.oauth2.authorizationserver.client.my-client-1.registration.redirect-
uris[1]=https://my-client-1.com/authorized
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[0]=openid
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[1]=profile
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[2]=email
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[3]=phone
spring.security.oauth2.authorizationserver.client.my-client-
1.registration.scopes[4]=address
spring.security.oauth2.authorizationserver.client.my-client-1.require-authorization-
consent=true
spring.security.oauth2.authorizationserver.client.my-client-2.registration.client-
id=efgh
spring.security.oauth2.authorizationserver.client.my-client-2.registration.client-
secret={noop}secret2
spring.security.oauth2.authorizationserver.client.my-client-2.registration.client-
authentication-methods[0]=client_secret_jwt
spring.security.oauth2.authorizationserver.client.my-client-
2.registration.authorization-grant-types[0]=client_credentials
spring.security.oauth2.authorizationserver.client.my-client-
2.registration.scopes[0]=user.read
spring.security.oauth2.authorizationserver.client.my-client-
2.registration.scopes[1]=user.write
spring.security.oauth2.authorizationserver.client.my-client-2.jwk-set-uri=https://my-
client-2.com/jwks
spring.security.oauth2.authorizationserver.client.my-client-2.token-endpoint-
authentication-signing-algorithm=RS256
```

Yaml

```
spring:
  security:
    oauth2:
      authorizationserver:
        client:
          my-client-1:
            registration:
              client-id: "abcd"
              client-secret: "{noop}secret1"
              client-authentication-methods:
                - "client_secret_basic"
              authorization-grant-types:
                - "authorization_code"
                - "refresh_token"
              redirect-uris:
                - "https://my-client-1.com/login/oauth2/code/abcd"
                - "https://my-client-1.com/authorized"
              scopes:
                - "openid"
                - "profile"
                - "email"
                - "phone"
                - "address"
            require-authorization-consent: true
          my-client-2:
            registration:
              client-id: "efgh"
              client-secret: "{noop}secret2"
              client-authentication-methods:
                - "client_secret_jwt"
              authorization-grant-types:
                - "client_credentials"
              scopes:
                - "user.read"
                - "user.write"
            jwk-set-uri: "https://my-client-2.com/jwks"
            token-endpoint-authentication-signing-algorithm: "RS256"
```

NOTE The `client-secret` property must be in a format that can be matched by the configured `PasswordEncoder`. The default instance of `PasswordEncoder` is created via `PasswordEncoderFactories.createDelegatingPasswordEncoder()`.

The auto-configuration Spring Boot provides for Spring Authorization Server is designed for getting started quickly. Most applications will require customization and will want to define several beans to override auto-configuration.

The following components can be defined as beans to override auto-configuration specific to Spring Authorization Server:

- `RegisteredClientRepository`
- `AuthorizationServerSettings`
- `SecurityFilterChain`
- `com.nimbusds.jose.jwk.source.JWKSource<com.nimbusds.jose.proc.SecurityContext>`
- `JwtDecoder`

TIP

Spring Boot auto-configures an `InMemoryRegisteredClientRepository` which is used by Spring Authorization Server for the management of registered clients. The `InMemoryRegisteredClientRepository` has limited capabilities and we recommend using it only for development environments. For production environments, consider using a `JdbcRegisteredClientRepository` or creating your own implementation of `RegisteredClientRepository`.

Additional information can be found in the Getting Started chapter of the Spring Authorization Server Reference Guide.

8.4.4. SAML 2.0

Relying Party

If you have `spring-security-saml2-service-provider` on your classpath, you can take advantage of some auto-configuration to set up a SAML 2.0 Relying Party. This configuration makes use of the properties under `Saml2RelyingPartyProperties`.

A relying party registration represents a paired configuration between an Identity Provider, IDP, and a Service Provider, SP. You can register multiple relying parties under the `spring.security.saml2.relyingparty` prefix, as shown in the following example:

Properties

```
spring.security.saml2.relyingparty.registration.my-relying-
party1.signing.credentials[0].private-key-location=path-to-private-key
spring.security.saml2.relyingparty.registration.my-relying-
party1.signing.credentials[0].certificate-location=path-to-certificate
spring.security.saml2.relyingparty.registration.my-relying-
party1.decryption.credentials[0].private-key-location=path-to-private-key
spring.security.saml2.relyingparty.registration.my-relying-
party1.decryption.credentials[0].certificate-location=path-to-certificate
spring.security.saml2.relyingparty.registration.my-relying-
party1.singlelogout.url=https://myapp/logout/saml2/slo
spring.security.saml2.relyingparty.registration.my-relying-
party1.singlelogout.response-url=https://remoteidp2.slo.url
spring.security.saml2.relyingparty.registration.my-relying-
party1.singlelogout.binding=POST
spring.security.saml2.relyingparty.registration.my-relying-
party1.assertingparty.verification.credentials[0].certificate-location=path-to-
verification-cert
spring.security.saml2.relyingparty.registration.my-relying-
party1.assertingparty.entity-id=remote-idp-entity-id1
spring.security.saml2.relyingparty.registration.my-relying-party1.assertingparty.sso-
url=https://remoteidp1.sso.url

spring.security.saml2.relyingparty.registration.my-relying-
party2.signing.credentials[0].private-key-location=path-to-private-key
spring.security.saml2.relyingparty.registration.my-relying-
party2.signing.credentials[0].certificate-location=path-to-certificate
spring.security.saml2.relyingparty.registration.my-relying-
party2.decryption.credentials[0].private-key-location=path-to-private-key
spring.security.saml2.relyingparty.registration.my-relying-
party2.decryption.credentials[0].certificate-location=path-to-certificate
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.verification.credentials[0].certificate-location=path-to-other-
verification-cert
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.entity-id=remote-idp-entity-id2
spring.security.saml2.relyingparty.registration.my-relying-party2.assertingparty.sso-
url=https://remoteidp2.sso.url
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.singlelogout.url=https://remoteidp2.slo.url
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.singlelogout.response-url=https://myapp/logout/saml2/slo
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.singlelogout.binding=POST
```

Yaml

```
spring:
  security:
    saml2:
      relyingparty:
        registration:
          my-relying-party1:
            signing:
              credentials:
              - private-key-location: "path-to-private-key"
                certificate-location: "path-to-certificate"
            decryption:
              credentials:
              - private-key-location: "path-to-private-key"
                certificate-location: "path-to-certificate"
            singlelogout:
               url: "https://myapp/logout/saml2/slo"
               response-url: "https://remoteidp2.slo.url"
               binding: "POST"
            assertingparty:
              verification:
                credentials:
                - certificate-location: "path-to-verification-cert"
              entity-id: "remote-idp-entity-id1"
              sso-url: "https://remoteidp1.sso.url"

          my-relying-party2:
            signing:
              credentials:
              - private-key-location: "path-to-private-key"
                certificate-location: "path-to-certificate"
            decryption:
              credentials:
              - private-key-location: "path-to-private-key"
                certificate-location: "path-to-certificate"
            assertingparty:
              verification:
                credentials:
                - certificate-location: "path-to-other-verification-cert"
              entity-id: "remote-idp-entity-id2"
              sso-url: "https://remoteidp2.sso.url"
              singlelogout:
                url: "https://remoteidp2.slo.url"
                response-url: "https://myapp/logout/saml2/slo"
                binding: "POST"
```

For SAML2 logout, by default, Spring Security's `Saml2LogoutRequestFilter` and `Saml2LogoutResponseFilter` only process URLs matching `/logout/saml2/slo`. If you want to customize the `url` to which AP-initiated logout requests get sent to or the `response-url` to which an AP sends

logout responses to, to use a different pattern, you need to provide configuration to process that custom pattern. For example, for servlet applications, you can add your own `SecurityFilterChain` that resembles the following:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration(proxyBeanMethods = false)
public class MySamlRelyingPartyConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http.authorizeHttpRequests((requests) ->
requests.anyRequest().authenticated());
        http.saml2Login(withDefaults());
        http.saml2Logout((saml2) -> saml2.logoutRequest((request) ->
request.logoutUrl("/SLOService.saml2"))
            .logoutResponse((response) -> response.logoutUrl("/SLOService.saml2")));
        return http.build();
    }

}
```

8.5. Spring Session

Spring Boot provides Spring Session auto-configuration for a wide range of data stores. When building a servlet web application, the following stores can be auto-configured:

- Redis
- JDBC
- Hazelcast
- MongoDB

Additionally, Spring Boot for Apache Geode provides auto-configuration for using Apache Geode as a session store.

The servlet auto-configuration replaces the need to use `@Enable*HttpSession`.

If a single Spring Session module is present on the classpath, Spring Boot uses that store implementation automatically. If you have more than one implementation, Spring Boot uses the following order for choosing a specific implementation:

1. Redis

2. JDBC
3. Hazelcast
4. MongoDB
5. If none of Redis, JDBC, Hazelcast and MongoDB are available, we do not configure a `SessionRepository`.

When building a reactive web application, the following stores can be auto-configured:

- Redis
- MongoDB

The reactive auto-configuration replaces the need to use `@Enable*WebSession`.

Similar to the servlet configuration, if you have more than one implementation, Spring Boot uses the following order for choosing a specific implementation:

1. Redis
2. MongoDB
3. If neither Redis nor MongoDB are available, we do not configure a `ReactiveSessionRepository`.

Each store has specific additional settings. For instance, it is possible to customize the name of the table for the JDBC store, as shown in the following example:

Properties

```
spring.session.jdbc.table-name=SESSIONS
```

Yaml

```
spring:
  session:
    jdbc:
      table-name: "SESSIONS"
```

For setting the timeout of the session you can use the `spring.session.timeout` property. If that property is not set with a servlet web application, the auto-configuration falls back to the value of `server.servlet.session.timeout`.

You can take control over Spring Session's configuration using `@Enable*HttpSession` (servlet) or `@Enable*WebSession` (reactive). This will cause the auto-configuration to back off. Spring Session can then be configured using the annotation's attributes rather than the previously described configuration properties.

8.6. Spring for GraphQL

If you want to build GraphQL applications, you can take advantage of Spring Boot's auto-configuration for Spring for GraphQL. The Spring for GraphQL project is based on GraphQL Java.

You'll need the `spring-boot-starter-graphql` starter at a minimum. Because GraphQL is transport-agnostic, you'll also need to have one or more additional starters in your application to expose your GraphQL API over the web:

Starter	Transport	Implementation
`spring-boot-starter-web`	HTTP	Spring MVC
`spring-boot-starter-websocket`	WebSocket	WebSocket for Servlet apps
`spring-boot-starter-webflux`	HTTP, WebSocket	Spring WebFlux
`spring-boot-starter-rsocket`	TCP, WebSocket	Spring WebFlux on Reactor Netty

8.6.1. GraphQL Schema

A Spring GraphQL application requires a defined schema at startup. By default, you can write ".graphqls" or ".gqls" schema files under `src/main/resources/graphql/**` and Spring Boot will pick them up automatically. You can customize the locations with `spring.graphql.schema.locations` and the file extensions with `spring.graphql.schema.file-extensions`.

NOTE	If you want Spring Boot to detect schema files in all your application modules and dependencies for that location, you can set `spring.graphql.schema.locations` to `"classpath*:graphql/**/"` (note the `classpath*:` prefix).

In the following sections, we'll consider this sample GraphQL schema, defining two types and two queries:

```
type Query {
    greeting(name: String! = "Spring"): String!
    project(slug: ID!): Project
}

""" A Project in the Spring portfolio """
type Project {
    """ Unique string id used in URLs """
    slug: ID!
    """ Project name """
    name: String!
    """ URL of the git repository """
    repositoryUrl: String!
    """ Current support status """
    status: ProjectStatus!
}

enum ProjectStatus {
    """ Actively supported by the Spring team """
    ACTIVE
    """ Supported by the community """
    COMMUNITY
    """ Prototype, not officially supported yet  """
    INCUBATING
    """ Project being retired, in maintenance mode """
    ATTIC
    """ End-Of-Lifed """
    EOL
}
```

NOTE By default, field introspection will be allowed on the schema as it is required for tools such as GraphiQL. If you wish to not expose information about the schema, you can disable introspection by setting `spring.graphql.schema.introspection.enabled` to `false`.

8.6.2. GraphQL RuntimeWiring

The GraphQL Java `RuntimeWiring.Builder` can be used to register custom scalar types, directives, type resolvers, `DataFetcher`, and more. You can declare `RuntimeWiringConfigurer` beans in your Spring config to get access to the `RuntimeWiring.Builder`. Spring Boot detects such beans and adds them to the GraphQlSource builder.

Typically, however, applications will not implement `DataFetcher` directly and will instead create annotated controllers. Spring Boot will automatically detect `@Controller` classes with annotated handler methods and register those as `DataFetcher`s. Here's a sample implementation for our greeting query with a `@Controller` class:

Java

```
import org.springframework.graphql.data.method.annotation.Argument;
import org.springframework.graphql.data.method.annotation.QueryMapping;
import org.springframework.stereotype.Controller;

@Controller
public class GreetingController {

    @QueryMapping
    public String greeting(@Argument String name) {
        return "Hello, " + name + "!";
    }

}
```

Kotlin

```
;

import org.springframework.graphql.data.method.annotation.Argument
import org.springframework.graphql.data.method.annotation.QueryMapping
import org.springframework.stereotype.Controller

@Controller
class GreetingController {

    @QueryMapping
    fun greeting(@Argument name: String): String {
        return "Hello, $name!"
    }

}
```

8.6.3. Querydsl and QueryByExample Repositories Support

Spring Data offers support for both Querydsl and QueryByExample repositories. Spring GraphQL can configure Querydsl and QueryByExample repositories as `DataFetcher`.

Spring Data repositories annotated with `@GraphQlRepository` and extending one of:

- `QuerydslPredicateExecutor`
- `ReactiveQuerydslPredicateExecutor`
- `QueryByExampleExecutor`
- `ReactiveQueryByExampleExecutor`

are detected by Spring Boot and considered as candidates for `DataFetcher` for matching top-level queries.

8.6.4. Transports

HTTP and WebSocket

The GraphQL HTTP endpoint is at HTTP POST `/graphql` by default. The path can be customized with `spring.graphql.path`.

TIP The HTTP endpoint for both Spring MVC and Spring WebFlux is provided by a `RouterFunction` bean with an `@Order` of `0`. If you define your own `RouterFunction` beans, you may want to add appropriate `@Order` annotations to ensure that they are sorted correctly.

The GraphQL WebSocket endpoint is off by default. To enable it:

- For a Servlet application, add the WebSocket starter `spring-boot-starter-websocket`
- For a WebFlux application, no additional dependency is required
- For both, the `spring.graphql.websocket.path` application property must be set

Spring GraphQL provides a Web Interception model. This is quite useful for retrieving information from an HTTP request header and set it in the GraphQL context or fetching information from the same context and writing it to a response header. With Spring Boot, you can declare a `WebInterceptor` bean to have it registered with the web transport.

Spring MVC and Spring WebFlux support CORS (Cross-Origin Resource Sharing) requests. CORS is a critical part of the web config for GraphQL applications that are accessed from browsers using different domains.

Spring Boot supports many configuration properties under the `spring.graphql.cors.*` namespace; here's a short configuration sample:

Properties

```
spring.graphql.cors.allowed-origins=https://example.org
spring.graphql.cors.allowed-methods=GET,POST
spring.graphql.cors.max-age=1800s
```

Yaml

```
spring:
  graphql:
    cors:
      allowed-origins: "https://example.org"
      allowed-methods: GET,POST
      max-age: 1800s
```

RSocket

RSocket is also supported as a transport, on top of WebSocket or TCP. Once the RSocket server is

configured, we can configure our GraphQL handler on a particular route using `spring.graphql.rsocket.mapping`. For example, configuring that mapping as `"graphql"` means we can use that as a route when sending requests with the `RSocketGraphQlClient`.

Spring Boot auto-configures a `RSocketGraphQlClient.Builder<?>` bean that you can inject in your components:

Java

```
@Component
public class RSocketGraphQlClientExample {

    private final RSocketGraphQlClient graphQlClient;

    public RSocketGraphQlClientExample(RSocketGraphQlClient.Builder<?> builder) {
        this.graphQlClient = builder.tcp("example.spring.io",
8181).route("graphql").build();
    }
```

Kotlin

```
@Component
class RSocketGraphQlClientExample(private val builder:
RSocketGraphQlClient.Builder<*>) {
```

And then send a request:

Java

```
Mono<Book> book = this.graphQlClient.document("{ bookById(id: \"book-1\"){ id name
pageCount author } }")
    .retrieve("bookById")
    .toEntity(Book.class);
```

Kotlin

```
val book = graphQlClient.document(
    """
    {
        bookById(id: "book-1"){
            id
            name
            pageCount
            author
        }
    }
    """
)
    .retrieve("bookById").toEntity(Book::class.java)
```

8.6.5. Exception Handling

Spring GraphQL enables applications to register one or more Spring `DataFetcherExceptionResolver` components that are invoked sequentially. The Exception must be resolved to a list of `graphql.GraphQLError` objects, see Spring GraphQL exception handling documentation. Spring Boot will automatically detect `DataFetcherExceptionResolver` beans and register them with the `GraphQlSource.Builder`.

8.6.6. GraphiQL and Schema printer

Spring GraphQL offers infrastructure for helping developers when consuming or developing a GraphQL API.

Spring GraphQL ships with a default GraphiQL page that is exposed at `"/graphiql"` by default. This page is disabled by default and can be turned on with the `spring.graphql.graphiql.enabled` property. Many applications exposing such a page will prefer a custom build. A default implementation is very useful during development, this is why it is exposed automatically with `spring-boot-devtools` during development.

You can also choose to expose the GraphQL schema in text format at `/graphql/schema` when the `spring.graphql.schema.printer.enabled` property is enabled.

8.7. Spring HATEOAS

If you develop a RESTful API that makes use of hypermedia, Spring Boot provides auto-configuration for Spring HATEOAS that works well with most applications. The auto-configuration replaces the need to use `@EnableHypermediaSupport` and registers a number of beans to ease building hypermedia-based applications, including a `LinkDiscoverers` (for client side support) and an `ObjectMapper` configured to correctly marshal responses into the desired representation. The `ObjectMapper` is customized by setting the various `spring.jackson.*` properties or, if one exists, by a `Jackson2ObjectMapperBuilder` bean.

You can take control of Spring HATEOAS's configuration by using `@EnableHypermediaSupport`. Note that doing so disables the `ObjectMapper` customization described earlier.

WARNING	`spring-boot-starter-hateoas` is specific to Spring MVC and should not be combined with Spring WebFlux. In order to use Spring HATEOAS with Spring WebFlux, you can add a direct dependency on `org.springframework.hateoas:spring-hateoas` along with `spring-boot-starter-webflux`.

By default, requests that accept `application/json` will receive an `application/hal+json` response. To disable this behavior set `spring.hateoas.use-hal-as-default-json-media-type` to `false` and define a `HypermediaMappingInformation` or `HalConfiguration` to configure Spring HATEOAS to meet the needs of your application and its clients.

8.8. What to Read Next

You should now have a good understanding of how to develop web applications with Spring Boot. The next few sections describe how Spring Boot integrates with various data technologies, messaging systems, and other IO capabilities. You can pick any of these based on your application's needs.

Chapter 9. Data

Spring Boot integrates with a number of data technologies, both SQL and NoSQL.

9.1. SQL Databases

The Spring Framework provides extensive support for working with SQL databases, from direct JDBC access using `JdbcTemplate` to complete "object relational mapping" technologies such as Hibernate. Spring Data provides an additional level of functionality: creating `Repository` implementations directly from interfaces and using conventions to generate queries from your method names.

9.1.1. Configure a DataSource

Java's `javax.sql.DataSource` interface provides a standard method of working with database connections. Traditionally, a `DataSource` uses a `URL` along with some credentials to establish a database connection.

TIP See the "How-to" section for more advanced examples, typically to take full control over the configuration of the DataSource.

Embedded Database Support

It is often convenient to develop applications by using an in-memory embedded database. Obviously, in-memory databases do not provide persistent storage. You need to populate your database when your application starts and be prepared to throw away data when your application ends.

TIP The "How-to" section includes a section on how to initialize a database.

Spring Boot can auto-configure embedded H2, HSQL, and Derby databases. You need not provide any connection URLs. You need only include a build dependency to the embedded database that you want to use. If there are multiple embedded databases on the classpath, set the `spring.datasource.embedded-database-connection` configuration property to control which one is used. Setting the property to `none` disables auto-configuration of an embedded database.

NOTE If you are using this feature in your tests, you may notice that the same database is reused by your whole test suite regardless of the number of application contexts that you use. If you want to make sure that each context has a separate embedded database, you should set `spring.datasource.generate-unique-name` to `true`.

For example, the typical POM dependencies would be as follows:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>runtime</scope>
</dependency>
```

NOTE You need a dependency on `spring-jdbc` for an embedded database to be auto-configured. In this example, it is pulled in transitively through `spring-boot-starter-data-jpa`.

TIP If, for whatever reason, you do configure the connection URL for an embedded database, take care to ensure that the database's automatic shutdown is disabled. If you use H2, you should use `DB_CLOSE_ON_EXIT=FALSE` to do so. If you use HSQLDB, you should ensure that `shutdown=true` is not used. Disabling the database's automatic shutdown lets Spring Boot control when the database is closed, thereby ensuring that it happens once access to the database is no longer needed.

Connection to a Production Database

Production database connections can also be auto-configured by using a pooling `DataSource`.

DataSource Configuration

DataSource configuration is controlled by external configuration properties in `spring.datasource.*`. For example, you might declare the following section in `application.properties`:

Properties

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
```

Yaml

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    password: "dbpass"
```

NOTE You should at least specify the URL by setting the `spring.datasource.url` property. Otherwise, Spring Boot tries to auto-configure an embedded database.

TIP	Spring Boot can deduce the JDBC driver class for most databases from the URL. If you need to specify a specific class, you can use the `spring.datasource.driver-class-name` property.
NOTE	For a pooling `DataSource` to be created, we need to be able to verify that a valid `Driver` class is available, so we check for that before doing anything. In other words, if you set `spring.datasource.driver-class-name=com.mysql.jdbc.Driver`, then that class has to be loadable.

See `DataSourceProperties` for more of the supported options. These are the standard options that work regardless of the actual implementation. It is also possible to fine-tune implementation-specific settings by using their respective prefix (`spring.datasource.hikari.*`, `spring.datasource.tomcat.*`, `spring.datasource.dbcp2.*`, and `spring.datasource.oracleucp.*`). See the documentation of the connection pool implementation you are using for more details.

For instance, if you use the Tomcat connection pool, you could customize many additional settings, as shown in the following example:

Properties

```
spring.datasource.tomcat.max-wait=10000
spring.datasource.tomcat.max-active=50
spring.datasource.tomcat.test-on-borrow=true
```

Yaml

```
spring:
  datasource:
    tomcat:
      max-wait: 10000
      max-active: 50
      test-on-borrow: true
```

This will set the pool to wait 10000ms before throwing an exception if no connection is available, limit the maximum number of connections to 50 and validate the connection before borrowing it from the pool.

Supported Connection Pools

Spring Boot uses the following algorithm for choosing a specific implementation:

1. We prefer HikariCP for its performance and concurrency. If HikariCP is available, we always choose it.
2. Otherwise, if the Tomcat pooling `DataSource` is available, we use it.
3. Otherwise, if Commons DBCP2 is available, we use it.
4. If none of HikariCP, Tomcat, and DBCP2 are available and if Oracle UCP is available, we use it.

NOTE	If you use the `spring-boot-starter-jdbc` or `spring-boot-starter-data-jpa` "starters", you automatically get a dependency to `HikariCP`.

You can bypass that algorithm completely and specify the connection pool to use by setting the `spring.datasource.type` property. This is especially important if you run your application in a Tomcat container, as `tomcat-jdbc` is provided by default.

Additional connection pools can always be configured manually, using `DataSourceBuilder`. If you define your own `DataSource` bean, auto-configuration does not occur. The following connection pools are supported by `DataSourceBuilder`:

- HikariCP
- Tomcat pooling `Datasource`
- Commons DBCP2
- Oracle UCP & `OracleDataSource`
- Spring Framework's `SimpleDriverDataSource`
- H2 `JdbcDataSource`
- PostgreSQL `PGSimpleDataSource`
- C3P0

Connection to a JNDI DataSource

If you deploy your Spring Boot application to an Application Server, you might want to configure and manage your DataSource by using your Application Server's built-in features and access it by using JNDI.

The `spring.datasource.jndi-name` property can be used as an alternative to the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties to access the `DataSource` from a specific JNDI location. For example, the following section in `application.properties` shows how you can access a JBoss AS defined `DataSource`:

Properties

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

Yaml

```
spring:
  datasource:
    jndi-name: "java:jboss/datasources/customers"
```

9.1.2. Using JdbcTemplate

Spring's `JdbcTemplate` and `NamedParameterJdbcTemplate` classes are auto-configured, and you can `@Autowire` them directly into your own beans, as shown in the following example:

Java

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

    public MyBean(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void doSomething() {
        this.jdbcTemplate ...
    }

}
```

Kotlin

```
import org.springframework.jdbc.core.JdbcTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val jdbcTemplate: JdbcTemplate) {

    fun doSomething() {
        jdbcTemplate.execute("delete from customer")
    }

}
```

You can customize some properties of the template by using the `spring.jdbc.template.*` properties, as shown in the following example:

Properties

```
spring.jdbc.template.max-rows=500
```

Yaml

```
spring:
  jdbc:
    template:
      max-rows: 500
```

NOTE	The `NamedParameterJdbcTemplate` reuses the same `JdbcTemplate` instance behind the scenes. If more than one `JdbcTemplate` is defined and no primary candidate exists, the `NamedParameterJdbcTemplate` is not auto-configured.

9.1.3. JPA and Spring Data JPA

The Java Persistence API is a standard technology that lets you "map" objects to relational databases. The `spring-boot-starter-data-jpa` POM provides a quick way to get started. It provides the following key dependencies:

- Hibernate: One of the most popular JPA implementations.
- Spring Data JPA: Helps you to implement JPA-based repositories.
- Spring ORM: Core ORM support from the Spring Framework.

TIP	We do not go into too many details of JPA or Spring Data here. You can follow the "Accessing Data with JPA" guide from spring.io and read the Spring Data JPA and Hibernate reference documentation.

Entity Classes

Traditionally, JPA "Entity" classes are specified in a `persistence.xml` file. With Spring Boot, this file is not necessary and "Entity Scanning" is used instead. By default the auto-configuration packages are scanned.

Any classes annotated with `@Entity`, `@Embeddable`, or `@MappedSuperclass` are considered. A typical entity class resembles the following example:

Java

```
import java.io.Serializable;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

@Entity
public class City implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String state;

    // ... additional members, often include @OneToMany mappings

    protected City() {
        // no-args constructor required by JPA spec
        // this one is protected since it should not be used directly
    }

    public City(String name, String state) {
        this.name = name;
        this.state = state;
    }

    public String getName() {
        return this.name;
    }

    public String getState() {
        return this.state;
    }

    // ... etc

}
```

Kotlin

```
import jakarta.persistence.Column
import jakarta.persistence.Entity
import jakarta.persistence.GeneratedValue
import jakarta.persistence.Id
import java.io.Serializable

@Entity
class City : Serializable {

    @Id
    @GeneratedValue
    private val id: Long? = null

    @Column(nullable = false)
    var name: String? = null
        private set

    // ... etc
    @Column(nullable = false)
    var state: String? = null
        private set

    // ... additional members, often include @OneToMany mappings

    protected constructor() {
        // no-args constructor required by JPA spec
        // this one is protected since it should not be used directly
    }

    constructor(name: String?, state: String?) {
        this.name = name
        this.state = state
    }

}
```

TIP You can customize entity scanning locations by using the `@EntityScan` annotation. See the "Separate @Entity Definitions from Spring Configuration" how-to.

Spring Data JPA Repositories

Spring Data JPA repositories are interfaces that you can define to access data. JPA queries are created automatically from your method names. For example, a `CityRepository` interface might declare a `findAllByState(String state)` method to find all the cities in a given state.

For more complex queries, you can annotate your method with Spring Data's `Query` annotation.

Spring Data repositories usually extend from the `Repository` or `CrudRepository` interfaces. If you use

auto-configuration, the auto-configuration packages are searched for repositories.

TIP	You can customize the locations to look for repositories using `@EnableJpaRepositories`.

The following example shows a typical Spring Data repository interface definition:

Java

```
import org.springframework.boot.docs.data.sql.jpaandspringdata.entityclasses.City;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.Repository;

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndStateAllIgnoringCase(String name, String state);

}
```

Kotlin

```
import org.springframework.boot.docs.data.sql.jpaandspringdata.entityclasses.City
import org.springframework.data.domain.Page
import org.springframework.data.domain.Pageable
import org.springframework.data.repository.Repository

interface CityRepository : Repository<City?, Long?> {

    fun findAll(pageable: Pageable?): Page<City?>?

    fun findByNameAndStateAllIgnoringCase(name: String?, state: String?): City?

}
```

Spring Data JPA repositories support three different modes of bootstrapping: default, deferred, and lazy. To enable deferred or lazy bootstrapping, set the `spring.data.jpa.repositories.bootstrap-mode` property to `deferred` or `lazy` respectively. When using deferred or lazy bootstrapping, the auto-configured `EntityManagerFactoryBuilder` will use the context's `AsyncTaskExecutor`, if any, as the bootstrap executor. If more than one exists, the one named `applicationTaskExecutor` will be used.

NOTE	When using deferred or lazy bootstrapping, make sure to defer any access to the JPA infrastructure after the application context bootstrap phase. You can use `SmartInitializingSingleton` to invoke any initialization that requires the JPA infrastructure. For JPA components (such as converters) that are created as Spring beans, use `ObjectProvider` to delay the resolution of dependencies, if any.

TIP	We have barely scratched the surface of Spring Data JPA. For complete details, see the Spring Data JPA reference documentation.

Spring Data Envers Repositories

If Spring Data Envers is available, JPA repositories are auto-configured to support typical Envers queries.

To use Spring Data Envers, make sure your repository extends from `RevisionRepository` as shown in the following example:

Java

```
import org.springframework.boot.docs.data.sql.jpaandspringdata.entityclasses.Country;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.history.RevisionRepository;

public interface CountryRepository extends RevisionRepository<Country, Long, Integer>,
Repository<Country, Long> {

    Page<Country> findAll(Pageable pageable);

}
```

Kotlin

```
import org.springframework.boot.docs.data.sql.jpaandspringdata.entityclasses.Country
import org.springframework.data.domain.Page
import org.springframework.data.domain.Pageable
import org.springframework.data.repository.Repository
import org.springframework.data.repository.history.RevisionRepository

interface CountryRepository :
        RevisionRepository<Country?, Long?, Int>,
        Repository<Country?, Long?> {

    fun findAll(pageable: Pageable?): Page<Country?>?

}
```

NOTE	For more details, check the Spring Data Envers reference documentation.

Creating and Dropping JPA Databases

By default, JPA databases are automatically created **only** if you use an embedded database (H2, HSQL, or Derby). You can explicitly configure JPA settings by using `spring.jpa.*` properties. For example, to create and drop tables you can add the following line to your `application.properties`:

Properties

```
spring.jpa.hibernate.ddl-auto=create-drop
```

Yaml

```
spring:
  jpa:
    hibernate.ddl-auto: "create-drop"
```

NOTE	Hibernate's own internal property name for this (if you happen to remember it better) is `hibernate.hbm2ddl.auto`. You can set it, along with other Hibernate native properties, by using `spring.jpa.properties.*` (the prefix is stripped before adding them to the entity manager). The following line shows an example of setting JPA properties for Hibernate:

Properties

```
spring.jpa.properties.hibernate[globally_quoted_identifiers]=true
```

Yaml

```
spring:
  jpa:
    properties:
      hibernate:
        "globally_quoted_identifiers": "true"
```

The line in the preceding example passes a value of `true` for the `hibernate.globally_quoted_identifiers` property to the Hibernate entity manager.

By default, the DDL execution (or validation) is deferred until the `ApplicationContext` has started. There is also a `spring.jpa.generate-ddl` flag, but it is not used if Hibernate auto-configuration is active, because the `ddl-auto` settings are more fine-grained.

Open EntityManager in View

If you are running a web application, Spring Boot by default registers `OpenEntityManagerInViewInterceptor` to apply the "Open EntityManager in View" pattern, to allow for lazy loading in web views. If you do not want this behavior, you should set `spring.jpa.open-in-view` to `false` in your `application.properties`.

9.1.4. Spring Data JDBC

Spring Data includes repository support for JDBC and will automatically generate SQL for the methods on `CrudRepository`. For more advanced queries, a `@Query` annotation is provided.

Spring Boot will auto-configure Spring Data's JDBC repositories when the necessary dependencies are on the classpath. They can be added to your project with a single dependency on `spring-boot-starter-data-jdbc`. If necessary, you can take control of Spring Data JDBC's configuration by adding the `@EnableJdbcRepositories` annotation or an `AbstractJdbcConfiguration` subclass to your application.

TIP	For complete details of Spring Data JDBC, see the reference documentation.

9.1.5. Using H2's Web Console

The H2 database provides a browser-based console that Spring Boot can auto-configure for you. The console is auto-configured when the following conditions are met:

- You are developing a servlet-based web application.
- `com.h2database:h2` is on the classpath.
- You are using Spring Boot's developer tools.

TIP	If you are not using Spring Boot's developer tools but would still like to make use of H2's console, you can configure the `spring.h2.console.enabled` property with a value of `true`.

NOTE	The H2 console is only intended for use during development, so you should take care to ensure that `spring.h2.console.enabled` is not set to `true` in production.

Changing the H2 Console's Path

By default, the console is available at `/h2-console`. You can customize the console's path by using the `spring.h2.console.path` property.

Accessing the H2 Console in a Secured Application

H2 Console uses frames and, as it is intended for development only, does not implement CSRF protection measures. If your application uses Spring Security, you need to configure it to

- disable CSRF protection for requests against the console,
- set the header `X-Frame-Options` to `SAMEORIGIN` on responses from the console.

More information on CSRF and the header X-Frame-Options can be found in the Spring Security Reference Guide.

In simple setups, a `SecurityFilterChain` like the following can be used:

Java

```
import org.springframework.boot.autoconfigure.security.servlet.PathRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Profile("dev")
@Configuration(proxyBeanMethods = false)
public class DevProfileSecurityConfiguration {

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    SecurityFilterChain h2ConsoleSecurityFilterChain(HttpSecurity http) throws
Exception {
        http.securityMatcher(PathRequest.toH2Console());
        http.authorizeHttpRequests(yourCustomAuthorization());
        http.csrf((csrf) -> csrf.disable());
        http.headers((headers) -> headers.frameOptions((frame) ->
frame.sameOrigin()));
        return http.build();
    }


}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Profile
import org.springframework.core.Ordered
import org.springframework.core.annotation.Order
import org.springframework.security.config.Customizer
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import org.springframework.security.web.SecurityFilterChain

@Profile("dev")
@Configuration(proxyBeanMethods = false)
class DevProfileSecurityConfiguration {

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    fun h2ConsoleSecurityFilterChain(http: HttpSecurity): SecurityFilterChain {
        return http.authorizeHttpRequests(yourCustomAuthorization())
            .csrf { csrf -> csrf.disable() }
            .headers { headers -> headers.frameOptions { frameOptions ->
frameOptions.sameOrigin() } }
            .build()
    }


}
```

WARNING	The H2 console is only intended for use during development. In production, disabling CSRF protection or allowing frames for a website may create severe security risks.

TIP	`PathRequest.toH2Console()` returns the correct request matcher also when the console's path has been customized.

9.1.6. Using jOOQ

jOOQ Object Oriented Querying (jOOQ) is a popular product from Data Geekery which generates Java code from your database and lets you build type-safe SQL queries through its fluent API. Both the commercial and open source editions can be used with Spring Boot.

Code Generation

In order to use jOOQ type-safe queries, you need to generate Java classes from your database schema. You can follow the instructions in the jOOQ user manual. If you use the `jooq-codegen-maven` plugin and you also use the `spring-boot-starter-parent` "parent POM", you can safely omit the plugin's `<version>` tag. You can also use Spring Boot-defined version variables (such as `h2.version`) to declare the plugin's database dependency. The following listing shows an example:

```
<plugin>
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <executions>
        ...
    </executions>
    <dependencies>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>${h2.version}</version>
        </dependency>
    </dependencies>
    <configuration>
        <jdbc>
            <driver>org.h2.Driver</driver>
            <url>jdbc:h2:~/yourdatabase</url>
        </jdbc>
        <generator>
            ...
        </generator>
    </configuration>
</plugin>
```

Using DSLContext

The fluent API offered by jOOQ is initiated through the `org.jooq.DSLContext` interface. Spring Boot auto-configures a `DSLContext` as a Spring Bean and connects it to your application `DataSource`. To use the `DSLContext`, you can inject it, as shown in the following example:

Java

```
import java.util.GregorianCalendar;
import java.util.List;

import org.jooq.DSLContext;

import org.springframework.stereotype.Component;

import static org.springframework.boot.docs.data.sql.jooq.dslcontext.Tables.AUTHOR;

@Component
public class MyBean {

    private final DSLContext create;

    public MyBean(DSLContext dslContext) {
        this.create = dslContext;
    }


}
```

Kotlin

```
import org.jooq.DSLContext
import org.springframework.stereotype.Component
import java.util.GregorianCalendar

@Component
class MyBean(private val create: DSLContext) {


}
```

TIP The jOOQ manual tends to use a variable named `create` to hold the `DSLContext`.

You can then use the `DSLContext` to construct your queries, as shown in the following example:

Java

```
public List<GregorianCalendar> authorsBornAfter1980() {
    return this.create.selectFrom(AUTHOR)
        .where(AUTHOR.DATE_OF_BIRTH.greaterThan(new GregorianCalendar(1980, 0, 1)))
        .fetch(AUTHOR.DATE_OF_BIRTH);
```

Kotlin

```
fun authorsBornAfter1980(): List<GregorianCalendar> {
    return create.selectFrom<Tables.TAuthorRecord>(Tables.AUTHOR)
        .where(Tables.AUTHOR?.DATE_OF_BIRTH?.greaterThan(GregorianCalendar(1980, 0,
1)))
        .fetch(Tables.AUTHOR?.DATE_OF_BIRTH)
}
```

jOOQ SQL Dialect

Unless the `spring.jooq.sql-dialect` property has been configured, Spring Boot determines the SQL dialect to use for your datasource. If Spring Boot could not detect the dialect, it uses `DEFAULT`.

NOTE	Spring Boot can only auto-configure dialects supported by the open source version of jOOQ.

Customizing jOOQ

More advanced customizations can be achieved by defining your own `DefaultConfigurationCustomizer` bean that will be invoked prior to creating the `org.jooq.Configuration @Bean`. This takes precedence to anything that is applied by the auto-configuration.

You can also create your own `org.jooq.Configuration @Bean` if you want to take complete control of the jOOQ configuration.

9.1.7. Using R2DBC

The Reactive Relational Database Connectivity (R2DBC) project brings reactive programming APIs to relational databases. R2DBC's `io.r2dbc.spi.Connection` provides a standard method of working with non-blocking database connections. Connections are provided by using a `ConnectionFactory`, similar to a `DataSource` with jdbc.

`ConnectionFactory` configuration is controlled by external configuration properties in `spring.r2dbc.*`. For example, you might declare the following section in `application.properties`:

Properties

```
spring.r2dbc.url=r2dbc:postgresql://localhost/test
spring.r2dbc.username=dbuser
spring.r2dbc.password=dbpass
```

Yaml

```
spring:
  r2dbc:
    url: "r2dbc:postgresql://localhost/test"
    username: "dbuser"
    password: "dbpass"
```

TIP	You do not need to specify a driver class name, since Spring Boot obtains the driver from R2DBC's Connection Factory discovery.

NOTE	At least the url should be provided. Information specified in the URL takes precedence over individual properties, that is `name`, `username`, `password` and pooling options.

TIP	The "How-to" section includes a section on how to initialize a database.

To customize the connections created by a `ConnectionFactory`, that is, set specific parameters that you do not want (or cannot) configure in your central database configuration, you can use a `ConnectionFactoryOptionsBuilderCustomizer` `@Bean`. The following example shows how to manually override the database port while the rest of the options are taken from the application configuration:

Java

```
import io.r2dbc.spi.ConnectionFactoryOptions;

import
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryOptionsBuilderCustomizer
;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyR2dbcConfiguration {

    @Bean
    public ConnectionFactoryOptionsBuilderCustomizer connectionFactoryPortCustomizer()
{
        return (builder) -> builder.option(ConnectionFactoryOptions.PORT, 5432);
    }

}
```

Kotlin

```
import io.r2dbc.spi.ConnectionFactoryOptions
import
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryOptionsBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyR2dbcConfiguration {

    @Bean
    fun connectionFactoryPortCustomizer(): ConnectionFactoryOptionsBuilderCustomizer {
        return ConnectionFactoryOptionsBuilderCustomizer { builder ->
            builder.option(ConnectionFactoryOptions.PORT, 5432)
        }
    }

}
```

The following examples show how to set some PostgreSQL connection options:

Java

```
import java.util.HashMap;
import java.util.Map;

import io.r2dbc.postgresql.PostgresqlConnectionFactoryProvider;

import
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryOptionsBuilderCustomizer
;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyPostgresR2dbcConfiguration {

    @Bean
    public ConnectionFactoryOptionsBuilderCustomizer postgresCustomizer() {
        Map<String, String> options = new HashMap<>();
        options.put("lock_timeout", "30s");
        options.put("statement_timeout", "60s");
        return (builder) ->
builder.option(PostgresqlConnectionFactoryProvider.OPTIONS, options);
    }

}
```

Kotlin

```
import io.r2dbc.postgresql.PostgresqlConnectionFactoryProvider
import
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryOptionsBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyPostgresR2dbcConfiguration {

    @Bean
    fun postgresCustomizer(): ConnectionFactoryOptionsBuilderCustomizer {
        val options: MutableMap<String, String> = HashMap()
        options["lock_timeout"] = "30s"
        options["statement_timeout"] = "60s"
        return ConnectionFactoryOptionsBuilderCustomizer { builder ->
            builder.option(PostgresqlConnectionFactoryProvider.OPTIONS, options)
        }
    }

}
```

When a `ConnectionFactory` bean is available, the regular JDBC `DataSource` auto-configuration backs off. If you want to retain the JDBC `DataSource` auto-configuration, and are comfortable with the risk of using the blocking JDBC API in a reactive application, add `@Import(DataSourceAutoConfiguration.class)` on a `@Configuration` class in your application to re-enable it.

Embedded Database Support

Similarly to the JDBC support, Spring Boot can automatically configure an embedded database for reactive usage. You need not provide any connection URLs. You need only include a build dependency to the embedded database that you want to use, as shown in the following example:

```
<dependency>
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

NOTE If you are using this feature in your tests, you may notice that the same database is reused by your whole test suite regardless of the number of application contexts that you use. If you want to make sure that each context has a separate embedded database, you should set `spring.r2dbc.generate-unique-name` to `true`.

Using DatabaseClient

A `DatabaseClient` bean is auto-configured, and you can `@Autowire` it directly into your own beans, as shown in the following example:

Java

```
import java.util.Map;

import reactor.core.publisher.Flux;

import org.springframework.r2dbc.core.DatabaseClient;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final DatabaseClient databaseClient;

    public MyBean(DatabaseClient databaseClient) {
        this.databaseClient = databaseClient;
    }

    public Flux<Map<String, Object>> someMethod() {
        return this.databaseClient.sql("select * from user").fetch().all();
    }

}
```

Kotlin

```
import org.springframework.r2dbc.core.DatabaseClient
import org.springframework.stereotype.Component
import reactor.core.publisher.Flux

@Component
class MyBean(private val databaseClient: DatabaseClient) {

    fun someMethod(): Flux<Map<String, Any>> {
        return databaseClient.sql("select * from user").fetch().all()
    }

}
```

Spring Data R2DBC Repositories

Spring Data R2DBC repositories are interfaces that you can define to access data. Queries are created automatically from your method names. For example, a `CityRepository` interface might declare a `findAllByState(String state)` method to find all the cities in a given state.

For more complex queries, you can annotate your method with Spring Data's `Query` annotation.

Spring Data repositories usually extend from the `Repository` or `CrudRepository` interfaces. If you use auto-configuration, the auto-configuration packages are searched for repositories.

The following example shows a typical Spring Data repository interface definition:

Java

```
import reactor.core.publisher.Mono;

import org.springframework.data.repository.Repository;

public interface CityRepository extends Repository<City, Long> {

    Mono<City> findByNameAndStateAllIgnoringCase(String name, String state);

}
```

Kotlin

```
import org.springframework.data.repository.Repository
import reactor.core.publisher.Mono

interface CityRepository : Repository<City?, Long?> {

    fun findByNameAndStateAllIgnoringCase(name: String?, state: String?): Mono<City?>?

}
```

TIP We have barely scratched the surface of Spring Data R2DBC. For complete details, see the Spring Data R2DBC reference documentation.

9.2. Working with NoSQL Technologies

Spring Data provides additional projects that help you access a variety of NoSQL technologies, including:

- Cassandra
- Couchbase
- Elasticsearch
- GemFire or Geode
- LDAP
- MongoDB
- Neo4J
- Redis

Of these, Spring Boot provides auto-configuration for Cassandra, Couchbase, Elasticsearch, LDAP, MongoDB, Neo4J and Redis. Additionally, Spring Boot for Apache Geode provides auto-configuration for Apache Geode. You can make use of the other projects, but you must configure them yourself. See the appropriate reference documentation at spring.io/projects/spring-data.

Spring Boot also provides auto-configuration for the InfluxDB client.

9.2.1. Redis

Redis is a cache, message broker, and richly-featured key-value store. Spring Boot offers basic auto-configuration for the Lettuce and Jedis client libraries and the abstractions on top of them provided by Spring Data Redis.

There is a `spring-boot-starter-data-redis` "Starter" for collecting the dependencies in a convenient way. By default, it uses Lettuce. That starter handles both traditional and reactive applications.

TIP	We also provide a `spring-boot-starter-data-redis-reactive` "Starter" for consistency with the other stores with reactive support.

Connecting to Redis

You can inject an auto-configured `RedisConnectionFactory`, `StringRedisTemplate`, or vanilla `RedisTemplate` instance as you would any other Spring Bean. The following listing shows an example of such a bean:

Java

```
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final StringRedisTemplate template;

    public MyBean(StringRedisTemplate template) {
        this.template = template;
    }

    public Boolean someMethod() {
        return this.template.hasKey("spring");
    }

}
```

Kotlin

```
import org.springframework.data.redis.core.StringRedisTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val template: StringRedisTemplate) {

    fun someMethod(): Boolean {
        return template.hasKey("spring")
    }

}
```

By default, the instance tries to connect to a Redis server at `localhost:6379`. You can specify custom connection details using `spring.data.redis.*` properties, as shown in the following example:

Properties

```
spring.data.redis.host=localhost
spring.data.redis.port=6379
spring.data.redis.database=0
spring.data.redis.username=user
spring.data.redis.password=secret
```

Yaml

```
spring:
  data:
    redis:
      host: "localhost"
      port: 6379
      database: 0
      username: "user"
      password: "secret"
```

TIP

You can also register an arbitrary number of beans that implement `LettuceClientConfigurationBuilderCustomizer` for more advanced customizations. `ClientResources` can also be customized using `ClientResourcesBuilderCustomizer`. If you use Jedis, `JedisClientConfigurationBuilderCustomizer` is also available. Alternatively, you can register a bean of type `RedisStandaloneConfiguration`, `RedisSentinelConfiguration`, or `RedisClusterConfiguration` to take full control over the configuration.

If you add your own `@Bean` of any of the auto-configured types, it replaces the default (except in the case of `RedisTemplate`, when the exclusion is based on the bean name, `redisTemplate`, not its type).

By default, a pooled connection factory is auto-configured if `commons-pool2` is on the classpath.

The auto-configured `RedisConnectionFactory` can be configured to use SSL for communication with the server by setting the properties as shown in this example:

Properties

```
spring.data.redis.ssl.enabled=true
```

Yaml

```
spring:
  data:
    redis:
      ssl:
        enabled: true
```

Custom SSL trust material can be configured in an SSL bundle and applied to the `RedisConnectionFactory` as shown in this example:

Properties

```
spring.data.redis.ssl.bundle=example
```

Yaml

```
spring:
  data:
    redis:
      ssl:
        bundle: "example"
```

9.2.2. MongoDB

MongoDB is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several conveniences for working with MongoDB, including the `spring-boot-starter-data-mongodb` and `spring-boot-starter-data-mongodb-reactive` "Starters".

Connecting to a MongoDB Database

To access MongoDB databases, you can inject an auto-configured `org.springframework.data.mongodb.MongoDatabaseFactory`. By default, the instance tries to connect to a MongoDB server at `mongodb://localhost/test`. The following example shows how to connect to a MongoDB database:

Java

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;

import org.springframework.data.mongodb.MongoDatabaseFactory;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final MongoDatabaseFactory mongo;

    public MyBean(MongoDatabaseFactory mongo) {
        this.mongo = mongo;
    }

    public MongoCollection<Document> someMethod() {
        MongoDatabase db = this.mongo.getMongoDatabase();
        return db.getCollection("users");
    }

}
```

Kotlin

```
import com.mongodb.client.MongoCollection
import org.bson.Document
import org.springframework.data.mongodb.MongoDatabaseFactory
import org.springframework.stereotype.Component

@Component
class MyBean(private val mongo: MongoDatabaseFactory) {

    fun someMethod(): MongoCollection<Document> {
        val db = mongo.mongoDatabase
        return db.getCollection("users")
    }

}
```

If you have defined your own `MongoClient`, it will be used to auto-configure a suitable `MongoDatabaseFactory`.

The auto-configured `MongoClient` is created using a `MongoClientSettings` bean. If you have defined your own `MongoClientSettings`, it will be used without modification and the `spring.data.mongodb` properties will be ignored. Otherwise a `MongoClientSettings` will be auto-configured and will have the `spring.data.mongodb` properties applied to it. In either case, you can declare one or more

`MongoClientSettingsBuilderCustomizer` beans to fine-tune the `MongoClientSettings` configuration. Each will be called in order with the `MongoClientSettings.Builder` that is used to build the `MongoClientSettings`.

You can set the `spring.data.mongodb.uri` property to change the URL and configure additional settings such as the *replica set*, as shown in the following example:

Properties

```
spring.data.mongodb.uri=mongodb://user:secret@mongoserver1.example.com:27017,mongoserv
er2.example.com:23456/test
```

Yaml

```
spring:
  data:
    mongodb:
      uri:
"mongodb://user:secret@mongoserver1.example.com:27017,mongoserver2.example.com:23456/t
est"
```

Alternatively, you can specify connection details using discrete properties. For example, you might declare the following settings in your `application.properties`:

Properties

```
spring.data.mongodb.host=mongoserver1.example.com
spring.data.mongodb.port=27017
spring.data.mongodb.additional-hosts[0]=mongoserver2.example.com:23456
spring.data.mongodb.database=test
spring.data.mongodb.username=user
spring.data.mongodb.password=secret
```

Yaml

```
spring:
  data:
    mongodb:
      host: "mongoserver1.example.com"
      port: 27017
      additional-hosts:
      - "mongoserver2.example.com:23456"
      database: "test"
      username: "user"
      password: "secret"
```

The auto-configured `MongoClient` can be configured to use SSL for communication with the server by setting the properties as shown in this example:

Properties

```
spring.data.mongodb.uri=mongodb://user:secret@mongoserver1.example.com:27017,mongoserv
er2.example.com:23456/test
spring.data.mongodb.ssl.enabled=true
```

Yaml

```
spring:
  data:
    mongodb:
      uri:
"mongodb://user:secret@mongoserver1.example.com:27017,mongoserver2.example.com:23456/t
est"
      ssl:
        enabled: true
```

Custom SSL trust material can be configured in an SSL bundle and applied to the `MongoClient` as shown in this example:

Properties

```
spring.data.mongodb.uri=mongodb://user:secret@mongoserver1.example.com:27017,mongoserv
er2.example.com:23456/test
spring.data.mongodb.ssl.bundle=example
```

Yaml

```
spring:
  data:
    mongodb:
      uri:
"mongodb://user:secret@mongoserver1.example.com:27017,mongoserver2.example.com:23456/t
est"
      ssl:
        bundle: "example"
```

TIP

If `spring.data.mongodb.port` is not specified, the default of `27017` is used. You could delete this line from the example shown earlier.

You can also specify the port as part of the host address by using the `host:port` syntax. This format should be used if you need to change the port of an `additional-hosts` entry.

TIP	If you do not use Spring Data MongoDB, you can inject a `MongoClient` bean instead of using `MongoDatabaseFactory`. If you want to take complete control of establishing the MongoDB connection, you can also declare your own `MongoDatabaseFactory` or `MongoClient` bean.
NOTE	If you are using the reactive driver, Netty is required for SSL. The auto-configuration configures this factory automatically if Netty is available and the factory to use has not been customized already.

MongoTemplate

Spring Data MongoDB provides a `MongoTemplate` class that is very similar in its design to Spring's `JdbcTemplate`. As with `JdbcTemplate`, Spring Boot auto-configures a bean for you to inject the template, as follows:

Java

```
import com.mongodb.client.MongoCollection;
import org.bson.Document;

import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final MongoTemplate mongoTemplate;

    public MyBean(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    public MongoCollection<Document> someMethod() {
        return this.mongoTemplate.getCollection("users");
    }

}
```

Kotlin

```
import com.mongodb.client.MongoCollection
import org.bson.Document
import org.springframework.data.mongodb.core.MongoTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val mongoTemplate: MongoTemplate) {

    fun someMethod(): MongoCollection<Document> {
        return mongoTemplate.getCollection("users")
    }

}
```

See the `MongoOperations` Javadoc for complete details.

Spring Data MongoDB Repositories

Spring Data includes repository support for MongoDB. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed automatically, based on method names.

In fact, both Spring Data JPA and Spring Data MongoDB share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a MongoDB data class rather than a JPA `@Entity`, it works in the same way, as shown in the following example:

Java

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.Repository;

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndStateAllIgnoringCase(String name, String state);

}
```

Kotlin

```
import org.springframework.data.domain.Page
import org.springframework.data.domain.Pageable
import org.springframework.data.repository.Repository

interface CityRepository :
    Repository<City?, Long?> {
    fun findAll(pageable: Pageable?): Page<City?>?
    fun findByNameAndStateAllIgnoringCase(name: String?, state: String?): City?
}
```

Repositories and documents are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and documents by using `@EnableMongoRepositories` and `@EntityScan` respectively.

TIP For complete details of Spring Data MongoDB, including its rich object mapping technologies, see its reference documentation.

9.2.3. Neo4j

Neo4j is an open-source NoSQL graph database that uses a rich data model of nodes connected by first class relationships, which is better suited for connected big data than traditional RDBMS approaches. Spring Boot offers several conveniences for working with Neo4j, including the `spring-boot-starter-data-neo4j` "Starter".

Connecting to a Neo4j Database

To access a Neo4j server, you can inject an auto-configured `org.neo4j.driver.Driver`. By default, the instance tries to connect to a Neo4j server at `localhost:7687` using the Bolt protocol. The following example shows how to inject a Neo4j `Driver` that gives you access, amongst other things, to a `Session`:

Java

```
import org.neo4j.driver.Driver;
import org.neo4j.driver.Session;
import org.neo4j.driver.Values;

import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final Driver driver;

    public MyBean(Driver driver) {
        this.driver = driver;
    }

    public String someMethod(String message) {
        try (Session session = this.driver.session()) {
            return session.executeWrite(
                    (transaction) -> transaction
                        .run("CREATE (a:Greeting) SET a.message = $message RETURN
a.message + ', from node ' + id(a)",
                                Values.parameters("message", message))
                        .single()
                        .get(0)
                        .asString());
        }
    }

}
```

Kotlin

```
import org.neo4j.driver.*
import org.springframework.stereotype.Component

@Component
class MyBean(private val driver: Driver) {
    fun someMethod(message: String?): String {
        driver.session().use { session ->
            return@someMethod session.executeWrite { transaction: TransactionContext
->
                transaction
                    .run(
                        "CREATE (a:Greeting) SET a.message = \$message RETURN
a.message + ', from node ' + id(a)",
                        Values.parameters("message", message)
                    )
                    .single()[0].asString()
            }
        }
    }
}
```

You can configure various aspects of the driver using `spring.neo4j.*` properties. The following example shows how to configure the uri and credentials to use:

Properties

```
spring.neo4j.uri=bolt://my-server:7687
spring.neo4j.authentication.username=neo4j
spring.neo4j.authentication.password=secret
```

Yaml

```
spring:
  neo4j:
    uri: "bolt://my-server:7687"
    authentication:
      username: "neo4j"
      password: "secret"
```

The auto-configured `Driver` is created using `ConfigBuilder`. To fine-tune its configuration, declare one or more `ConfigBuilderCustomizer` beans. Each will be called in order with the `ConfigBuilder` that is used to build the `Driver`.

Spring Data Neo4j Repositories

Spring Data includes repository support for Neo4j. For complete details of Spring Data Neo4j, see the reference documentation.

Spring Data Neo4j shares the common infrastructure with Spring Data JPA as many other Spring Data modules do. You could take the JPA example from earlier and define `City` as Spring Data Neo4j `@Node` rather than JPA `@Entity` and the repository abstraction works in the same way, as shown in the following example:

Java

```
import java.util.Optional;

import org.springframework.data.neo4j.repository.Neo4jRepository;

public interface CityRepository extends Neo4jRepository<City, Long> {

    Optional<City> findOneByNameAndState(String name, String state);

}
```

Kotlin

```
import org.springframework.data.neo4j.repository.Neo4jRepository
import java.util.Optional

interface CityRepository : Neo4jRepository<City?, Long?> {

    fun findOneByNameAndState(name: String?, state: String?): Optional<City?>?

}
```

The `spring-boot-starter-data-neo4j` "Starter" enables the repository support as well as transaction management. Spring Boot supports both classic and reactive Neo4j repositories, using the `Neo4jTemplate` or `ReactiveNeo4jTemplate` beans. When Project Reactor is available on the classpath, the reactive style is also auto-configured.

Repositories and entities are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and entities by using `@EnableNeo4jRepositories` and `@EntityScan` respectively.

NOTE

In an application using the reactive style, a `ReactiveTransactionManager` is not auto-configured. To enable transaction management, the following bean must be defined in your configuration:

Java

```
import org.neo4j.driver.Driver;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.data.neo4j.core.ReactiveDatabaseSelectionProvider;
import
org.springframework.data.neo4j.core.transaction.ReactiveNeo4jTransaction
Manager;

@Configuration(proxyBeanMethods = false)
public class MyNeo4jConfiguration {

    @Bean
    public ReactiveNeo4jTransactionManager
reactiveTransactionManager(Driver driver,
            ReactiveDatabaseSelectionProvider databaseNameProvider) {
        return new ReactiveNeo4jTransactionManager(driver,
databaseNameProvider);
    }

}
```

Kotlin

```
import org.neo4j.driver.Driver
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import
org.springframework.data.neo4j.core.ReactiveDatabaseSelectionProvider
import
org.springframework.data.neo4j.core.transaction.ReactiveNeo4jTransaction
Manager

@Configuration(proxyBeanMethods = false)
class MyNeo4jConfiguration {

    @Bean
    fun reactiveTransactionManager(driver: Driver,
            databaseNameProvider: ReactiveDatabaseSelectionProvider):
ReactiveNeo4jTransactionManager {
        return ReactiveNeo4jTransactionManager(driver,
databaseNameProvider)
    }
}
```

9.2.4. Elasticsearch

Elasticsearch is an open source, distributed, RESTful search and analytics engine. Spring Boot offers basic auto-configuration for Elasticsearch clients.

Spring Boot supports several clients:

- The official low-level REST client
- The official Java API client
- The `ReactiveElasticsearchClient` provided by Spring Data Elasticsearch

Spring Boot provides a dedicated "Starter", `spring-boot-starter-data-elasticsearch`.

Connecting to Elasticsearch Using REST clients

Elasticsearch ships two different REST clients that you can use to query a cluster: the low-level client from the `org.elasticsearch.client:elasticsearch-rest-client` module and the Java API client from the `co.elastic.clients:elasticsearch-java` module. Additionally, Spring Boot provides support for a reactive client from the `org.springframework.data:spring-data-elasticsearch` module. By default, the clients will target `localhost:9200`. You can use `spring.elasticsearch.*` properties to further tune how the clients are configured, as shown in the following example:

Properties

```
spring.elasticsearch.uris=https://search.example.com:9200
spring.elasticsearch.socket-timeout=10s
spring.elasticsearch.username=user
spring.elasticsearch.password=secret
```

Yaml

```
spring:
  elasticsearch:
    uris: "https://search.example.com:9200"
    socket-timeout: "10s"
    username: "user"
    password: "secret"
```

Connecting to Elasticsearch Using RestClient

If you have `elasticsearch-rest-client` on the classpath, Spring Boot will auto-configure and register a `RestClient` bean. In addition to the properties described previously, to fine-tune the `RestClient` you can register an arbitrary number of beans that implement `RestClientBuilderCustomizer` for more advanced customizations. To take full control over the clients' configuration, define a `RestClientBuilder` bean.

Additionally, if `elasticsearch-rest-client-sniffer` is on the classpath, a `Sniffer` is auto-configured to automatically discover nodes from a running Elasticsearch cluster and set them on the `RestClient` bean. You can further tune how `Sniffer` is configured, as shown in the following

example:

Properties

```
spring.elasticsearch.restclient.sniffer.interval=10m
spring.elasticsearch.restclient.sniffer.delay-after-failure=30s
```

Yaml

```
spring:
  elasticsearch:
    restclient:
      sniffer:
        interval: "10m"
        delay-after-failure: "30s"
```

Connecting to Elasticsearch Using ElasticsearchClient

If you have `co.elastic.clients:elasticsearch-java` on the classpath, Spring Boot will auto-configure and register an `ElasticsearchClient` bean.

The `ElasticsearchClient` uses a transport that depends upon the previously described `RestClient`. Therefore, the properties described previously can be used to configure the `ElasticsearchClient`. Furthermore, you can define a `TransportOptions` bean to take further control of the behavior of the transport.

Connecting to Elasticsearch using ReactiveElasticsearchClient

Spring Data Elasticsearch ships `ReactiveElasticsearchClient` for querying Elasticsearch instances in a reactive fashion. If you have Spring Data Elasticsearch and Reactor on the classpath, Spring Boot will auto-configure and register a `ReactiveElasticsearchClient`.

The `ReactiveElasticsearchclient` uses a transport that depends upon the previously described `RestClient`. Therefore, the properties described previously can be used to configure the `ReactiveElasticsearchClient`. Furthermore, you can define a `TransportOptions` bean to take further control of the behavior of the transport.

Connecting to Elasticsearch by Using Spring Data

To connect to Elasticsearch, an `ElasticsearchClient` bean must be defined, auto-configured by Spring Boot or manually provided by the application (see previous sections). With this configuration in place, an `ElasticsearchTemplate` can be injected like any other Spring bean, as shown in the following example:

Java

```
import org.springframework.data.elasticsearch.client.elc.ElasticsearchTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final ElasticsearchTemplate template;

    public MyBean(ElasticsearchTemplate template) {
        this.template = template;
    }

    public boolean someMethod(String id) {
        return this.template.exists(id, User.class);
    }

}
```

Kotlin

```
import org.springframework.stereotype.Component

@Component
class MyBean(private val template:
org.springframework.data.elasticsearch.client.erhlc.ElasticsearchRestTemplate ) {

    fun someMethod(id: String): Boolean {
        return template.exists(id, User::class.java)
    }

}
```

In the presence of `spring-data-elasticsearch` and Reactor, Spring Boot can also auto-configure a ReactiveElasticsearchClient and a `ReactiveElasticsearchTemplate` as beans. They are the reactive equivalent of the other REST clients.

Spring Data Elasticsearch Repositories

Spring Data includes repository support for Elasticsearch. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed for you automatically based on method names.

In fact, both Spring Data JPA and Spring Data Elasticsearch share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now an Elasticsearch `@Document` class rather than a JPA `@Entity`, it works in the same way.

Repositories and documents are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and documents by

using `@EnableElasticsearchRepositories` and `@EntityScan` respectively.

TIP	For complete details of Spring Data Elasticsearch, see the reference documentation.

Spring Boot supports both classic and reactive Elasticsearch repositories, using the `ElasticsearchRestTemplate` or `ReactiveElasticsearchTemplate` beans. Most likely those beans are auto-configured by Spring Boot given the required dependencies are present.

If you wish to use your own template for backing the Elasticsearch repositories, you can add your own `ElasticsearchRestTemplate` or `ElasticsearchOperations` `@Bean`, as long as it is named `"elasticsearchTemplate"`. Same applies to `ReactiveElasticsearchTemplate` and `ReactiveElasticsearchOperations`, with the bean name `"reactiveElasticsearchTemplate"`.

You can choose to disable the repositories support with the following property:

Properties

```
spring.data.elasticsearch.repositories.enabled=false
```

Yaml

```
spring:
  data:
    elasticsearch:
      repositories:
        enabled: false
```

9.2.5. Cassandra

Cassandra is an open source, distributed database management system designed to handle large amounts of data across many commodity servers. Spring Boot offers auto-configuration for Cassandra and the abstractions on top of it provided by Spring Data Cassandra. There is a `spring-boot-starter-data-cassandra` “Starter” for collecting the dependencies in a convenient way.

Connecting to Cassandra

You can inject an auto-configured `CassandraTemplate` or a Cassandra `CqlSession` instance as you would with any other Spring Bean. The `spring.cassandra.*` properties can be used to customize the connection. Generally, you provide `keyspace-name` and `contact-points` as well the local datacenter name, as shown in the following example:

Properties

```
spring.cassandra.keyspace-name=mykeyspace
spring.cassandra.contact-points=cassandrahost1:9042,cassandrahost2:9042
spring.cassandra.local-datacenter=datacenter1
```

Yaml

```
spring:
  cassandra:
    keyspace-name: "mykeyspace"
    contact-points: "cassandrahost1:9042,cassandrahost2:9042"
    local-datacenter: "datacenter1"
```

If the port is the same for all your contact points you can use a shortcut and only specify the host names, as shown in the following example:

Properties

```
spring.cassandra.keyspace-name=mykeyspace
spring.cassandra.contact-points=cassandrahost1,cassandrahost2
spring.cassandra.local-datacenter=datacenter1
```

Yaml

```
spring:
  cassandra:
    keyspace-name: "mykeyspace"
    contact-points: "cassandrahost1,cassandrahost2"
    local-datacenter: "datacenter1"
```

TIP Those two examples are identical as the port default to `9042`. If you need to configure the port, use `spring.cassandra.port`.

The auto-configured `CqlSession` can be configured to use SSL for communication with the server by setting the properties as shown in this example:

Properties

```
spring.cassandra.keyspace-name=mykeyspace
spring.cassandra.contact-points=cassandrahost1,cassandrahost2
spring.cassandra.local-datacenter=datacenter1
spring.cassandra.ssl.enabled=true
```

Yaml

```
spring:
  cassandra:
    keyspace-name: "mykeyspace"
    contact-points: "cassandrahost1,cassandrahost2"
    local-datacenter: "datacenter1"
    ssl:
      enabled: true
```

Custom SSL trust material can be configured in an SSL bundle and applied to the `CqlSession` as shown in this example:

Properties

```
spring.cassandra.keyspace-name=mykeyspace
spring.cassandra.contact-points=cassandrahost1,cassandrahost2
spring.cassandra.local-datacenter=datacenter1
spring.cassandra.ssl.bundle=example
```

Yaml

```
spring:
  cassandra:
    keyspace-name: "mykeyspace"
    contact-points: "cassandrahost1,cassandrahost2"
    local-datacenter: "datacenter1"
    ssl:
      bundle: "example"
```

NOTE

The Cassandra driver has its own configuration infrastructure that loads an `application.conf` at the root of the classpath.

Spring Boot does not look for such a file by default but can load one using `spring.cassandra.config`. If a property is both present in `spring.cassandra.*` and the configuration file, the value in `spring.cassandra.*` takes precedence.

For more advanced driver customizations, you can register an arbitrary number of beans that implement `DriverConfigLoaderBuilderCustomizer`. The `CqlSession` can be customized with a bean of type `CqlSessionBuilderCustomizer`.

NOTE

If you use `CqlSessionBuilder` to create multiple `CqlSession` beans, keep in mind the builder is mutable so make sure to inject a fresh copy for each session.

The following code listing shows how to inject a Cassandra bean:

Java

```
import org.springframework.data.cassandra.core.CassandraTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final CassandraTemplate template;

    public MyBean(CassandraTemplate template) {
        this.template = template;
    }

    public long someMethod() {
        return this.template.count(User.class);
    }

}
```

Kotlin

```
import org.springframework.data.cassandra.core.CassandraTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val template: CassandraTemplate) {

    fun someMethod(): Long {
        return template.count(User::class.java)
    }

}
```

If you add your own `@Bean` of type `CassandraTemplate`, it replaces the default.

Spring Data Cassandra Repositories

Spring Data includes basic repository support for Cassandra. Currently, this is more limited than the JPA repositories discussed earlier and needs `@Query` annotated finder methods.

Repositories and entities are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and entities by using `@EnableCassandraRepositories` and `@EntityScan` respectively.

TIP For complete details of Spring Data Cassandra, see the reference documentation.

9.2.6. Couchbase

Couchbase is an open-source, distributed, multi-model NoSQL document-oriented database that is optimized for interactive applications. Spring Boot offers auto-configuration for Couchbase and the abstractions on top of it provided by Spring Data Couchbase. There are `spring-boot-starter-data-couchbase` and `spring-boot-starter-data-couchbase-reactive` "Starters" for collecting the dependencies in a convenient way.

Connecting to Couchbase

You can get a `Cluster` by adding the Couchbase SDK and some configuration. The `spring.couchbase.*` properties can be used to customize the connection. Generally, you provide the connection string, username, and password, as shown in the following example:

Properties

```
spring.couchbase.connection-string=couchbase://192.168.1.123
spring.couchbase.username=user
spring.couchbase.password=secret
```

Yaml

```
spring:
  couchbase:
    connection-string: "couchbase://192.168.1.123"
    username: "user"
    password: "secret"
```

It is also possible to customize some of the `ClusterEnvironment` settings. For instance, the following configuration changes the timeout to open a new `Bucket` and enables SSL support with a reference to a configured SSL bundle:

Properties

```
spring.couchbase.env.timeouts.connect=3s
spring.couchbase.env.ssl.bundle=example
```

Yaml

```
spring:
  couchbase:
    env:
      timeouts:
        connect: "3s"
      ssl:
        bundle: "example"
```

TIP	Check the `spring.couchbase.env.*` properties for more details. To take more control, one or more `ClusterEnvironmentBuilderCustomizer` beans can be used.

Spring Data Couchbase Repositories

Spring Data includes repository support for Couchbase.

Repositories and documents are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and documents by using `@EnableCouchbaseRepositories` and `@EntityScan` respectively.

For complete details of Spring Data Couchbase, see the reference documentation.

You can inject an auto-configured `CouchbaseTemplate` instance as you would with any other Spring Bean, provided a `CouchbaseClientFactory` bean is available. This happens when a `Cluster` is available, as described above, and a bucket name has been specified:

Properties

```
spring.data.couchbase.bucket-name=my-bucket
```

Yaml

```
spring:
  data:
    couchbase:
      bucket-name: "my-bucket"
```

The following examples shows how to inject a `CouchbaseTemplate` bean:

Java

```
import org.springframework.data.couchbase.core.CouchbaseTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final CouchbaseTemplate template;

    public MyBean(CouchbaseTemplate template) {
        this.template = template;
    }

    public String someMethod() {
        return this.template.getBucketName();
    }

}
```

Kotlin

```
import org.springframework.data.couchbase.core.CouchbaseTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val template: CouchbaseTemplate) {

    fun someMethod(): String {
        return template.bucketName
    }

}
```

There are a few beans that you can define in your own configuration to override those provided by the auto-configuration:

- A `CouchbaseMappingContext` `@Bean` with a name of `couchbaseMappingContext`.
- A `CustomConversions` `@Bean` with a name of `couchbaseCustomConversions`.
- A `CouchbaseTemplate` `@Bean` with a name of `couchbaseTemplate`.

To avoid hard-coding those names in your own config, you can reuse `BeanNames` provided by Spring Data Couchbase. For instance, you can customize the converters to use, as follows:

Java

```
import org.assertj.core.util.Arrays;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.couchbase.config.BeanNames;
import org.springframework.data.couchbase.core.convert.CouchbaseCustomConversions;

@Configuration(proxyBeanMethods = false)
public class MyCouchbaseConfiguration {

    @Bean(BeanNames.COUCHBASE_CUSTOM_CONVERSIONS)
    public CouchbaseCustomConversions myCustomConversions() {
        return new CouchbaseCustomConversions(Arrays.asList(new MyConverter()));
    }

}
```

Kotlin

```
import org.assertj.core.util.Arrays
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.data.couchbase.config.BeanNames
import org.springframework.data.couchbase.core.convert.CouchbaseCustomConversions

@Configuration(proxyBeanMethods = false)
class MyCouchbaseConfiguration {

    @Bean(BeanNames.COUCHBASE_CUSTOM_CONVERSIONS)
    fun myCustomConversions(): CouchbaseCustomConversions {
        return CouchbaseCustomConversions(Arrays.asList(MyConverter()))
    }

}
```

9.2.7. LDAP

LDAP (Lightweight Directory Access Protocol) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an IP network. Spring Boot offers auto-configuration for any compliant LDAP server as well as support for the embedded in-memory LDAP server from UnboundID.

LDAP abstractions are provided by Spring Data LDAP. There is a `spring-boot-starter-data-ldap` "Starter" for collecting the dependencies in a convenient way.

Connecting to an LDAP Server

To connect to an LDAP server, make sure you declare a dependency on the `spring-boot-starter-data-ldap` "Starter" or `spring-ldap-core` and then declare the URLs of your server in your application.properties, as shown in the following example:

Properties

```
spring.ldap.urls=ldap://myserver:1235
spring.ldap.username=admin
spring.ldap.password=secret
```

Yaml

```
spring:
  ldap:
    urls: "ldap://myserver:1235"
    username: "admin"
    password: "secret"
```

If you need to customize connection settings, you can use the `spring.ldap.base` and

`spring.ldap.base-environment` properties.

An `LdapContextSource` is auto-configured based on these settings. If a `DirContextAuthenticationStrategy` bean is available, it is associated to the auto-configured `LdapContextSource`. If you need to customize it, for instance to use a `PooledContextSource`, you can still inject the auto-configured `LdapContextSource`. Make sure to flag your customized `ContextSource` as `@Primary` so that the auto-configured `LdapTemplate` uses it.

Spring Data LDAP Repositories

Spring Data includes repository support for LDAP.

Repositories and documents are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and documents by using `@EnableLdapRepositories` and `@EntityScan` respectively.

For complete details of Spring Data LDAP, see the reference documentation.

You can also inject an auto-configured `LdapTemplate` instance as you would with any other Spring Bean, as shown in the following example:

Java

```
import java.util.List;

import org.springframework.ldap.core.LdapTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final LdapTemplate template;

    public MyBean(LdapTemplate template) {
        this.template = template;
    }

    public List<User> someMethod() {
        return this.template.findAll(User.class);
    }

}
```

Kotlin

```
import org.springframework.ldap.core.LdapTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val template: LdapTemplate) {

    fun someMethod(): List<User> {
        return template.findAll(User::class.java)
    }

}
```

Embedded In-memory LDAP Server

For testing purposes, Spring Boot supports auto-configuration of an in-memory LDAP server from UnboundID. To configure the server, add a dependency to `com.unboundid:unboundid-ldapsdk` and declare a `spring.ldap.embedded.base-dn` property, as follows:

Properties

```
spring.ldap.embedded.base-dn=dc=spring,dc=io
```

Yaml

```
spring:
  ldap:
    embedded:
      base-dn: "dc=spring,dc=io"
```

NOTE

It is possible to define multiple base-dn values, however, since distinguished names usually contain commas, they must be defined using the correct notation.

In yaml files, you can use the yaml list notation. In properties files, you must include the index as part of the property name:

Properties

```
spring.ldap.embedded.base-dn[0]=dc=spring,dc=io
spring.ldap.embedded.base-dn[1]=dc=vmware,dc=com
```

Yaml

```
spring.ldap.embedded.base-dn:
  - "dc=spring,dc=io"
  - "dc=vmware,dc=com"
```

By default, the server starts on a random port and triggers the regular LDAP support. There is no need to specify a `spring.ldap.urls` property.

If there is a `schema.ldif` file on your classpath, it is used to initialize the server. If you want to load the initialization script from a different resource, you can also use the `spring.ldap.embedded.ldif` property.

By default, a standard schema is used to validate `LDIF` files. You can turn off validation altogether by setting the `spring.ldap.embedded.validation.enabled` property. If you have custom attributes, you can use `spring.ldap.embedded.validation.schema` to define your custom attribute types or object classes.

9.2.8. InfluxDB

InfluxDB is an open-source time series database optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet-of-Things sensor data, and real-time analytics.

Connecting to InfluxDB

Spring Boot auto-configures an `InfluxDB` instance, provided the `influxdb-java` client is on the classpath and the URL of the database is set, as shown in the following example:

Properties

```
spring.influx.url=https://172.0.0.1:8086
```

Yaml

```
spring:
  influx:
    url: "https://172.0.0.1:8086"
```

If the connection to InfluxDB requires a user and password, you can set the `spring.influx.user` and `spring.influx.password` properties accordingly.

InfluxDB relies on OkHttp. If you need to tune the http client `InfluxDB` uses behind the scenes, you can register an `InfluxDbOkHttpClientBuilderProvider` bean.

If you need more control over the configuration, consider registering an `InfluxDbCustomizer` bean.

9.3. What to Read Next

You should now have a feeling for how to use Spring Boot with various data technologies. From here, you can read about Spring Boot's support for various messaging technologies and how to enable them in your application.

Chapter 10. Messaging

The Spring Framework provides extensive support for integrating with messaging systems, from simplified use of the JMS API using `JmsTemplate` to a complete infrastructure to receive messages asynchronously. Spring AMQP provides a similar feature set for the Advanced Message Queuing Protocol. Spring Boot also provides auto-configuration options for `RabbitTemplate` and RabbitMQ. Spring WebSocket natively includes support for STOMP messaging, and Spring Boot has support for that through starters and a small amount of auto-configuration. Spring Boot also has support for Apache Kafka.

10.1. JMS

The `jakarta.jms.ConnectionFactory` interface provides a standard method of creating a `jakarta.jms.Connection` for interacting with a JMS broker. Although Spring needs a `ConnectionFactory` to work with JMS, you generally need not use it directly yourself and can instead rely on higher level messaging abstractions. (See the relevant section of the Spring Framework reference documentation for details.) Spring Boot also auto-configures the necessary infrastructure to send and receive messages.

10.1.1. ActiveMQ "Classic" Support

When ActiveMQ "Classic" is available on the classpath, Spring Boot can configure a `ConnectionFactory`.

NOTE	If you use `spring-boot-starter-activemq`, the necessary dependencies to connect to an ActiveMQ "Classic" instance are provided, as is the Spring infrastructure to integrate with JMS.

ActiveMQ "Classic" configuration is controlled by external configuration properties in `spring.activemq.*`. By default, ActiveMQ "Classic" is auto-configured to use the TCP transport, connecting by default to `tcp://localhost:61616`. The following example shows how to change the default broker URL:

Properties

```
spring.activemq.broker-url=tcp://192.168.1.210:9876
spring.activemq.user=admin
spring.activemq.password=secret
```

Yaml

```
spring:
  activemq:
    broker-url: "tcp://192.168.1.210:9876"
    user: "admin"
    password: "secret"
```

By default, a `CachingConnectionFactory` wraps the native `ConnectionFactory` with sensible settings that you can control by external configuration properties in `spring.jms.*`:

Properties

```
spring.jms.cache.session-cache-size=5
```

Yaml

```
spring:
  jms:
    cache:
      session-cache-size: 5
```

If you'd rather use native pooling, you can do so by adding a dependency to `org.messaginghub:pooled-jms` and configuring the `JmsPoolConnectionFactory` accordingly, as shown in the following example:

Properties

```
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=50
```

Yaml

```
spring:
  activemq:
    pool:
      enabled: true
      max-connections: 50
```

TIP See `ActiveMQProperties` for more of the supported options. You can also register an arbitrary number of beans that implement `ActiveMQConnectionFactoryCustomizer` for more advanced customizations.

By default, ActiveMQ "Classic" creates a destination if it does not yet exist so that destinations are resolved against their provided names.

10.1.2. ActiveMQ Artemis Support

Spring Boot can auto-configure a `ConnectionFactory` when it detects that ActiveMQ Artemis is available on the classpath. If the broker is present, an embedded broker is automatically started and configured (unless the mode property has been explicitly set). The supported modes are `embedded` (to make explicit that an embedded broker is required and that an error should occur if the broker is not available on the classpath) and `native` (to connect to a broker using the `netty` transport protocol). When the latter is configured, Spring Boot configures a `ConnectionFactory` that connects to a broker running on the local machine with the default settings.

NOTE	If you use `spring-boot-starter-artemis`, the necessary dependencies to connect to an existing ActiveMQ Artemis instance are provided, as well as the Spring infrastructure to integrate with JMS. Adding `org.apache.activemq:artemis-jakarta-server` to your application lets you use embedded mode.

ActiveMQ Artemis configuration is controlled by external configuration properties in `spring.artemis.*`. For example, you might declare the following section in `application.properties`:

Properties

```
spring.artemis.mode=native
spring.artemis.broker-url=tcp://192.168.1.210:9876
spring.artemis.user=admin
spring.artemis.password=secret
```

Yaml

```
spring:
  artemis:
    mode: native
    broker-url: "tcp://192.168.1.210:9876"
    user: "admin"
    password: "secret"
```

When embedding the broker, you can choose if you want to enable persistence and list the destinations that should be made available. These can be specified as a comma-separated list to create them with the default options, or you can define bean(s) of type `org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration` or `org.apache.activemq.artemis.jms.server.config.TopicConfiguration`, for advanced queue and topic configurations, respectively.

By default, a `CachingConnectionFactory` wraps the native `ConnectionFactory` with sensible settings that you can control by external configuration properties in `spring.jms.*`:

Properties

```
spring.jms.cache.session-cache-size=5
```

Yaml

```
spring:
  jms:
    cache:
      session-cache-size: 5
```

If you'd rather use native pooling, you can do so by adding a dependency on `org.messaginghub:pooled-jms` and configuring the `JmsPoolConnectionFactory` accordingly, as shown in the following example:

Properties

```
spring.artemis.pool.enabled=true
spring.artemis.pool.max-connections=50
```

Yaml

```
spring:
  artemis:
    pool:
      enabled: true
      max-connections: 50
```

See `ArtemisProperties` for more supported options.

No JNDI lookup is involved, and destinations are resolved against their names, using either the `name` attribute in the ActiveMQ Artemis configuration or the names provided through configuration.

10.1.3. Using a JNDI ConnectionFactory

If you are running your application in an application server, Spring Boot tries to locate a JMS `ConnectionFactory` by using JNDI. By default, the `java:/JmsXA` and `java:/XAConnectionFactory` location are checked. You can use the `spring.jms.jndi-name` property if you need to specify an alternative location, as shown in the following example:

Properties

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

Yaml

```
spring:
  jms:
    jndi-name: "java:/MyConnectionFactory"
```

10.1.4. Sending a Message

Spring's `JmsTemplate` is auto-configured, and you can autowire it directly into your own beans, as shown in the following example:

Java

```
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JmsTemplate jmsTemplate;

    public MyBean(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

    public void someMethod() {
        this.jmsTemplate.convertAndSend("hello");
    }

}
```

Kotlin

```
import org.springframework.jms.core.JmsTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val jmsTemplate: JmsTemplate) {

    fun someMethod() {
        jmsTemplate.convertAndSend("hello")
    }

}
```

NOTE `JmsMessagingTemplate` can be injected in a similar manner. If a `DestinationResolver` or a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `JmsTemplate`.

10.1.5. Receiving a Message

When the JMS infrastructure is present, any bean can be annotated with `@JmsListener` to create a listener endpoint. If no `JmsListenerContainerFactory` has been defined, a default one is configured automatically. If a `DestinationResolver`, a `MessageConverter`, or a `jakarta.jms.ExceptionListener` beans are defined, they are associated automatically with the default factory.

By default, the default factory is transactional. If you run in an infrastructure where a `JtaTransactionManager` is present, it is associated to the listener container by default. If not, the `sessionTransacted` flag is enabled. In that latter scenario, you can associate your local data store transaction to the processing of an incoming message by adding `@Transactional` on your listener

method (or a delegate thereof). This ensures that the incoming message is acknowledged, once the local transaction has completed. This also includes sending response messages that have been performed on the same JMS session.

The following component creates a listener endpoint on the `someQueue` destination:

Java

```
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @JmsListener(destination = "someQueue")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.jms.annotation.JmsListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @JmsListener(destination = "someQueue")
    fun processMessage(content: String?) {
        // ...
    }

}
```

TIP See the Javadoc of `@EnableJms` for more details.

If you need to create more `JmsListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `DefaultJmsListenerContainerFactoryConfigurer` that you can use to initialize a `DefaultJmsListenerContainerFactory` with the same settings as the one that is auto-configured.

For instance, the following example exposes another factory that uses a specific `MessageConverter`:

Java

```
import jakarta.jms.ConnectionFactory;

import
org.springframework.boot.autoconfigure.jms.DefaultJmsListenerContainerFactoryConfigure
r;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;

@Configuration(proxyBeanMethods = false)
public class MyJmsConfiguration {

    @Bean
    public DefaultJmsListenerContainerFactory
myFactory(DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
        ConnectionFactory connectionFactory = getCustomConnectionFactory();
        configurer.configure(factory, connectionFactory);
        factory.setMessageConverter(new MyMessageConverter());
        return factory;
    }

    private ConnectionFactory getCustomConnectionFactory() {
        return ...
    }

}
```

Kotlin

```
import jakarta.jms.ConnectionFactory
import
org.springframework.boot.autoconfigure.jms.DefaultJmsListenerContainerFactoryConfigure
r
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.jms.config.DefaultJmsListenerContainerFactory

@Configuration(proxyBeanMethods = false)
class MyJmsConfiguration {

    @Bean
    fun myFactory(configurer: DefaultJmsListenerContainerFactoryConfigurer):
DefaultJmsListenerContainerFactory {
        val factory = DefaultJmsListenerContainerFactory()
        val connectionFactory = getCustomConnectionFactory()
        configurer.configure(factory, connectionFactory)
        factory.setMessageConverter(MyMessageConverter())
        return factory
    }

    fun getCustomConnectionFactory() : ConnectionFactory? {
        return ...
    }

}
```

Then you can use the factory in any `@JmsListener`-annotated method as follows:

Java

```
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @JmsListener(destination = "someQueue", containerFactory = "myFactory")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.jms.annotation.JmsListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @JmsListener(destination = "someQueue", containerFactory = "myFactory")
    fun processMessage(content: String?) {
        // ...
    }

}
```

10.2. AMQP

The Advanced Message Queuing Protocol (AMQP) is a platform-neutral, wire-level protocol for message-oriented middleware. The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions. Spring Boot offers several conveniences for working with AMQP through RabbitMQ, including the `spring-boot-starter-amqp` "Starter".

10.2.1. RabbitMQ Support

RabbitMQ is a lightweight, reliable, scalable, and portable message broker based on the AMQP protocol. Spring uses RabbitMQ to communicate through the AMQP protocol.

RabbitMQ configuration is controlled by external configuration properties in `spring.rabbitmq.*`. For example, you might declare the following section in `application.properties`:

Properties

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

Yaml

```
spring:
  rabbitmq:
    host: "localhost"
    port: 5672
    username: "admin"
    password: "secret"
```

Alternatively, you could configure the same connection using the `addresses` attribute:

Properties

```
spring.rabbitmq.addresses=amqp://admin:secret@localhost
```

Yaml

```
spring:
  rabbitmq:
    addresses: "amqp://admin:secret@localhost"
```

NOTE	When specifying addresses that way, the `host` and `port` properties are ignored. If the address uses the `amqps` protocol, SSL support is enabled automatically.

See `RabbitProperties` for more of the supported property-based configuration options. To configure lower-level details of the RabbitMQ `ConnectionFactory` that is used by Spring AMQP, define a `ConnectionFactoryCustomizer` bean.

If a `ConnectionNameStrategy` bean exists in the context, it will be automatically used to name connections created by the auto-configured `CachingConnectionFactory`.

To make an application-wide, additive customization to the `RabbitTemplate`, use a `RabbitTemplateCustomizer` bean.

TIP	See Understanding AMQP, the protocol used by RabbitMQ for more details.

10.2.2. Sending a Message

Spring's `AmqpTemplate` and `AmqpAdmin` are auto-configured, and you can autowire them directly into your own beans, as shown in the following example:

Java

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final AmqpAdmin amqpAdmin;

    private final AmqpTemplate amqpTemplate;

    public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
        this.amqpAdmin = amqpAdmin;
        this.amqpTemplate = amqpTemplate;
    }

    public void someMethod() {
        this.amqpAdmin.getQueueInfo("someQueue");
    }

    public void someOtherMethod() {
        this.amqpTemplate.convertAndSend("hello");
    }

}
```

Kotlin

```
import org.springframework.amqp.core.AmqpAdmin
import org.springframework.amqp.core.AmqpTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val amqpAdmin: AmqpAdmin, private val amqpTemplate: AmqpTemplate)
{

    fun someMethod() {
        amqpAdmin.getQueueInfo("someQueue")
    }

    fun someOtherMethod() {
        amqpTemplate.convertAndSend("hello")
    }

}
```

NOTE	`RabbitMessagingTemplate` can be injected in a similar manner. If a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `AmqpTemplate`.

If necessary, any `org.springframework.amqp.core.Queue` that is defined as a bean is automatically used to declare a corresponding queue on the RabbitMQ instance.

To retry operations, you can enable retries on the `AmqpTemplate` (for example, in the event that the broker connection is lost):

Properties

```
spring.rabbitmq.template.retry.enabled=true
spring.rabbitmq.template.retry.initial-interval=2s
```

Yaml

```
spring:
  rabbitmq:
    template:
      retry:
        enabled: true
        initial-interval: "2s"
```

Retries are disabled by default. You can also customize the `RetryTemplate` programmatically by declaring a `RabbitRetryTemplateCustomizer` bean.

If you need to create more `RabbitTemplate` instances or if you want to override the default, Spring Boot provides a `RabbitTemplateConfigurer` bean that you can use to initialize a `RabbitTemplate` with the same settings as the factories used by the auto-configuration.

10.2.3. Sending a Message To A Stream

To send a message to a particular stream, specify the name of the stream, as shown in the following example:

Properties

```
spring.rabbitmq.stream.name=my-stream
```

Yaml

```
spring:
  rabbitmq:
    stream:
      name: "my-stream"
```

If a `MessageConverter`, `StreamMessageConverter`, or `ProducerCustomizer` bean is defined, it is associated automatically to the auto-configured `RabbitStreamTemplate`.

If you need to create more `RabbitStreamTemplate` instances or if you want to override the default, Spring Boot provides a `RabbitStreamTemplateConfigurer` bean that you can use to initialize a `RabbitStreamTemplate` with the same settings as the factories used by the auto-configuration.

10.2.4. Receiving a Message

When the Rabbit infrastructure is present, any bean can be annotated with `@RabbitListener` to create a listener endpoint. If no `RabbitListenerContainerFactory` has been defined, a default `SimpleRabbitListenerContainerFactory` is automatically configured and you can switch to a direct container using the `spring.rabbitmq.listener.type` property. If a `MessageConverter` or a `MessageRecoverer` bean is defined, it is automatically associated with the default factory.

The following sample component creates a listener endpoint on the `someQueue` queue:

Java

```
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @RabbitListener(queues = "someQueue")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.amqp.rabbit.annotation.RabbitListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @RabbitListener(queues = ["someQueue"])
    fun processMessage(content: String?) {
        // ...
    }

}
```

TIP See the Javadoc of `@EnableRabbit` for more details.

If you need to create more `RabbitListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `SimpleRabbitListenerContainerFactoryConfigurer` and a `DirectRabbitListenerContainerFactoryConfigurer` that you can use to initialize a

`SimpleRabbitListenerContainerFactory` and a `DirectRabbitListenerContainerFactory` with the same settings as the factories used by the auto-configuration.

TIP	It does not matter which container type you chose. Those two beans are exposed by the auto-configuration.

For instance, the following configuration class exposes another factory that uses a specific `MessageConverter`:

Java

```
import org.springframework.amqp.rabbit.config.SimpleRabbitListenerContainerFactory;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import
org.springframework.boot.autoconfigure.amqp.SimpleRabbitListenerContainerFactoryConfig
urer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyRabbitConfiguration {

    @Bean
    public SimpleRabbitListenerContainerFactory
myFactory(SimpleRabbitListenerContainerFactoryConfigurer configurer) {
        SimpleRabbitListenerContainerFactory factory = new
SimpleRabbitListenerContainerFactory();
        ConnectionFactory connectionFactory = getCustomConnectionFactory();
        configurer.configure(factory, connectionFactory);
        factory.setMessageConverter(new MyMessageConverter());
        return factory;
    }

    private ConnectionFactory getCustomConnectionFactory() {
        return ...
    }

}
```

Kotlin

```
import org.springframework.amqp.rabbit.config.SimpleRabbitListenerContainerFactory
import org.springframework.amqp.rabbit.connection.ConnectionFactory
import
org.springframework.boot.autoconfigure.amqp.SimpleRabbitListenerContainerFactoryConfig
urer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyRabbitConfiguration {

    @Bean
    fun myFactory(configurer: SimpleRabbitListenerContainerFactoryConfigurer):
SimpleRabbitListenerContainerFactory {
        val factory = SimpleRabbitListenerContainerFactory()
        val connectionFactory = getCustomConnectionFactory()
        configurer.configure(factory, connectionFactory)
        factory.setMessageConverter(MyMessageConverter())
        return factory
    }

    fun getCustomConnectionFactory() : ConnectionFactory? {
        return ...
    }

}
```

Then you can use the factory in any `@RabbitListener`-annotated method, as follows:

Java

```
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @RabbitListener(queues = "someQueue", containerFactory = "myFactory")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.amqp.rabbit.annotation.RabbitListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @RabbitListener(queues = ["someQueue"], containerFactory = "myFactory")
    fun processMessage(content: String?) {
        // ...
    }

}
```

You can enable retries to handle situations where your listener throws an exception. By default, `RejectAndDontRequeueRecoverer` is used, but you can define a `MessageRecoverer` of your own. When retries are exhausted, the message is rejected and either dropped or routed to a dead-letter exchange if the broker is configured to do so. By default, retries are disabled. You can also customize the `RetryTemplate` programmatically by declaring a `RabbitRetryTemplateCustomizer` bean.

IMPORTANT

By default, if retries are disabled and the listener throws an exception, the delivery is retried indefinitely. You can modify this behavior in two ways: Set the `defaultRequeueRejected` property to `false` so that zero re-deliveries are attempted or throw an `AmqpRejectAndDontRequeueException` to signal the message should be rejected. The latter is the mechanism used when retries are enabled and the maximum number of delivery attempts is reached.

10.3. Apache Kafka Support

Apache Kafka is supported by providing auto-configuration of the `spring-kafka` project.

Kafka configuration is controlled by external configuration properties in `spring.kafka.*`. For example, you might declare the following section in `application.properties`:

Properties

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=myGroup
```

Yaml

```
spring:
  kafka:
    bootstrap-servers: "localhost:9092"
    consumer:
      group-id: "myGroup"
```

TIP	To create a topic on startup, add a bean of type `NewTopic`. If the topic already exists, the bean is ignored.

See `KafkaProperties` for more supported options.

10.3.1. Sending a Message

Spring's `KafkaTemplate` is auto-configured, and you can autowire it directly in your own beans, as shown in the following example:

Java

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public MyBean(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void someMethod() {
        this.kafkaTemplate.send("someTopic", "Hello");
    }

}
```

Kotlin

```
import org.springframework.kafka.core.KafkaTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val kafkaTemplate: KafkaTemplate<String, String>) {

    fun someMethod() {
        kafkaTemplate.send("someTopic", "Hello")
    }

}
```

NOTE	If the property `spring.kafka.producer.transaction-id-prefix` is defined, a `KafkaTransactionManager` is automatically configured. Also, if a `RecordMessageConverter` bean is defined, it is automatically associated to the auto-configured `KafkaTemplate`.

10.3.2. Receiving a Message

When the Apache Kafka infrastructure is present, any bean can be annotated with `@KafkaListener` to create a listener endpoint. If no `KafkaListenerContainerFactory` has been defined, a default one is automatically configured with keys defined in `spring.kafka.listener.*`.

The following component creates a listener endpoint on the `someTopic` topic:

Java

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @KafkaListener(topics = "someTopic")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.kafka.annotation.KafkaListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @KafkaListener(topics = ["someTopic"])
    fun processMessage(content: String?) {
        // ...
    }

}
```

If a `KafkaTransactionManager` bean is defined, it is automatically associated to the container factory. Similarly, if a `RecordFilterStrategy`, `CommonErrorHandler`, `AfterRollbackProcessor` or `ConsumerAwareRebalanceListener` bean is defined, it is automatically associated to the default factory.

Depending on the listener type, a `RecordMessageConverter` or `BatchMessageConverter` bean is associated to the default factory. If only a `RecordMessageConverter` bean is present for a batch listener, it is wrapped in a `BatchMessageConverter`.

TIP A custom `ChainedKafkaTransactionManager` must be marked `@Primary` as it usually references the auto-configured `KafkaTransactionManager` bean.

10.3.3. Kafka Streams

Spring for Apache Kafka provides a factory bean to create a `StreamsBuilder` object and manage the lifecycle of its streams. Spring Boot auto-configures the required `KafkaStreamsConfiguration` bean as long as `kafka-streams` is on the classpath and Kafka Streams is enabled by the `@EnableKafkaStreams` annotation.

Enabling Kafka Streams means that the application id and bootstrap servers must be set. The former can be configured using `spring.kafka.streams.application-id`, defaulting to `spring.application.name` if not set. The latter can be set globally or specifically overridden only for streams.

Several additional properties are available using dedicated properties; other arbitrary Kafka properties can be set using the `spring.kafka.streams.properties` namespace. See also Additional Kafka Properties for more information.

To use the factory bean, wire `StreamsBuilder` into your `@Bean` as shown in the following example:

Java

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Produced;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafkaStreams;
import org.springframework.kafka.support.serializer.JsonSerde;

@Configuration(proxyBeanMethods = false)
@EnableKafkaStreams
public class MyKafkaStreamsConfiguration {

    @Bean
    public KStream<Integer, String> kStream(StreamsBuilder streamsBuilder) {
        KStream<Integer, String> stream = streamsBuilder.stream("ks1In");
        stream.map(this::uppercaseValue).to("ks1Out", Produced.with(Serdes.Integer(),
new JsonSerde<>()));
        return stream;
    }

    private KeyValue<Integer, String> uppercaseValue(Integer key, String value) {
        return new KeyValue<>(key, value.toUpperCase());
    }

}
```

Kotlin

```
import org.apache.kafka.common.serialization.Serdes
import org.apache.kafka.streams.KeyValue
import org.apache.kafka.streams.StreamsBuilder
import org.apache.kafka.streams.kstream.KStream
import org.apache.kafka.streams.kstream.Produced
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.kafka.annotation.EnableKafkaStreams
import org.springframework.kafka.support.serializer.JsonSerde

@Configuration(proxyBeanMethods = false)
@EnableKafkaStreams
class MyKafkaStreamsConfiguration {

    @Bean
    fun kStream(streamsBuilder: StreamsBuilder): KStream<Int, String> {
        val stream = streamsBuilder.stream<Int, String>("ks1In")
        stream.map(this::uppercaseValue).to("ks1Out", Produced.with(Serdes.Integer(),
JsonSerde()))
        return stream
    }

    private fun uppercaseValue(key: Int, value: String): KeyValue<Int?, String?> {
        return KeyValue(key, value.uppercase())
    }

}
```

By default, the streams managed by the `StreamBuilder` object are started automatically. You can customize this behavior using the `spring.kafka.streams.auto-startup` property.

10.3.4. Additional Kafka Properties

The properties supported by auto configuration are shown in the "Integration Properties" section of the Appendix. Note that, for the most part, these properties (hyphenated or camelCase) map directly to the Apache Kafka dotted properties. See the Apache Kafka documentation for details.

Properties that don't include a client type (`producer`, `consumer`, `admin`, or `streams`) in their name are considered to be common and apply to all clients. Most of these common properties can be overridden for one or more of the client types, if needed.

Apache Kafka designates properties with an importance of HIGH, MEDIUM, or LOW. Spring Boot auto-configuration supports all HIGH importance properties, some selected MEDIUM and LOW properties, and any properties that do not have a default value.

Only a subset of the properties supported by Kafka are available directly through the `KafkaProperties` class. If you wish to configure the individual client types with additional properties that are not directly supported, use the following properties:

Properties

```
spring.kafka.properties[prop.one]=first
spring.kafka.admin.properties[prop.two]=second
spring.kafka.consumer.properties[prop.three]=third
spring.kafka.producer.properties[prop.four]=fourth
spring.kafka.streams.properties[prop.five]=fifth
```

Yaml

```
spring:
  kafka:
    properties:
      "[prop.one]": "first"
    admin:
      properties:
        "[prop.two]": "second"
    consumer:
      properties:
        "[prop.three]": "third"
    producer:
      properties:
        "[prop.four]": "fourth"
    streams:
      properties:
        "[prop.five]": "fifth"
```

This sets the common `prop.one` Kafka property to `first` (applies to producers, consumers, admins, and streams), the `prop.two` admin property to `second`, the `prop.three` consumer property to `third`, the `prop.four` producer property to `fourth` and the `prop.five` streams property to `fifth`.

You can also configure the Spring Kafka `JsonDeserializer` as follows:

Properties

```
spring.kafka.consumer.value-
deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties[spring.json.value.default.type]=com.example.Invoice
spring.kafka.consumer.properties[spring.json.trusted.packages]=com.example.main,com.ex
ample.another
```

Yaml

```
spring:
  kafka:
    consumer:
      value-deserializer:
"org.springframework.kafka.support.serializer.JsonDeserializer"
      properties:
        "[spring.json.value.default.type]": "com.example.Invoice"
        "[spring.json.trusted.packages]": "com.example.main,com.example.another"
```

Similarly, you can disable the `JsonSerializer` default behavior of sending type information in headers:

Properties

```
spring.kafka.producer.value-
serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties[spring.json.add.type.headers]=false
```

Yaml

```
spring:
  kafka:
    producer:
      value-serializer: "org.springframework.kafka.support.serializer.JsonSerializer"
      properties:
        "[spring.json.add.type.headers]": false
```

IMPORTANT	Properties set in this way override any configuration item that Spring Boot explicitly supports.

10.3.5. Testing with Embedded Kafka

Spring for Apache Kafka provides a convenient way to test projects with an embedded Apache Kafka broker. To use this feature, annotate a test class with `@EmbeddedKafka` from the `spring-kafka-test` module. For more information, please see the Spring for Apache Kafka reference manual.

To make Spring Boot auto-configuration work with the aforementioned embedded Apache Kafka broker, you need to remap a system property for embedded broker addresses (populated by the `EmbeddedKafkaBroker`) into the Spring Boot configuration property for Apache Kafka. There are several ways to do that:

- Provide a system property to map embedded broker addresses into `spring.kafka.bootstrap-servers` in the test class:

Java

```
static {
    System.setProperty(EmbeddedKafkaBroker.BROKER_LIST_PROPERTY,
"spring.kafka.bootstrap-servers");
}
```

Kotlin

```
init {
    System.setProperty(EmbeddedKafkaBroker.BROKER_LIST_PROPERTY,
"spring.kafka.bootstrap-servers")
}
```

- Configure a property name on the `@EmbeddedKafka` annotation:

Java

```
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.kafka.test.context.EmbeddedKafka;

@SpringBootTest
@EmbeddedKafka(topics = "someTopic", bootstrapServersProperty =
"spring.kafka.bootstrap-servers")
class MyTest {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.kafka.test.context.EmbeddedKafka

@SpringBootTest
@EmbeddedKafka(topics = ["someTopic"], bootstrapServersProperty =
"spring.kafka.bootstrap-servers")
class MyTest {

    // ...

}
```

- Use a placeholder in configuration properties:

Properties

```
spring.kafka.bootstrap-servers=${spring.embedded.kafka.brokers}
```

Yaml

```
spring:
  kafka:
    bootstrap-servers: "${spring.embedded.kafka.brokers}"
```

10.4. RSocket

RSocket is a binary protocol for use on byte stream transports. It enables symmetric interaction models through async message passing over a single connection.

The `spring-messaging` module of the Spring Framework provides support for RSocket requesters and responders, both on the client and on the server side. See the RSocket section of the Spring Framework reference for more details, including an overview of the RSocket protocol.

10.4.1. RSocket Strategies Auto-configuration

Spring Boot auto-configures an `RSocketStrategies` bean that provides all the required infrastructure for encoding and decoding RSocket payloads. By default, the auto-configuration will try to configure the following (in order):

1. CBOR codecs with Jackson
2. JSON codecs with Jackson

The `spring-boot-starter-rsocket` starter provides both dependencies. See the Jackson support section to know more about customization possibilities.

Developers can customize the `RSocketStrategies` component by creating beans that implement the `RSocketStrategiesCustomizer` interface. Note that their `@Order` is important, as it determines the order of codecs.

10.4.2. RSocket server Auto-configuration

Spring Boot provides RSocket server auto-configuration. The required dependencies are provided by the `spring-boot-starter-rsocket`.

Spring Boot allows exposing RSocket over WebSocket from a WebFlux server, or standing up an independent RSocket server. This depends on the type of application and its configuration.

For WebFlux application (that is of type `WebApplicationType.REACTIVE`), the RSocket server will be plugged into the Web Server only if the following properties match:

Properties

```
spring.rsocket.server.mapping-path=/rsocket
spring.rsocket.server.transport=websocket
```

Yaml

```
spring:
  rsocket:
    server:
      mapping-path: "/rsocket"
      transport: "websocket"
```

WARNING	Plugging RSocket into a web server is only supported with Reactor Netty, as RSocket itself is built with that library.

Alternatively, an RSocket TCP or websocket server is started as an independent, embedded server. Besides the dependency requirements, the only required configuration is to define a port for that server:

Properties

```
spring.rsocket.server.port=9898
```

Yaml

```
spring:
  rsocket:
    server:
      port: 9898
```

10.4.3. Spring Messaging RSocket support

Spring Boot will auto-configure the Spring Messaging infrastructure for RSocket.

This means that Spring Boot will create a `RSocketMessageHandler` bean that will handle RSocket requests to your application.

10.4.4. Calling RSocket Services with RSocketRequester

Once the `RSocket` channel is established between server and client, any party can send or receive requests to the other.

As a server, you can get injected with an `RSocketRequester` instance on any handler method of an RSocket `@Controller`. As a client, you need to configure and establish an RSocket connection first. Spring Boot auto-configures an `RSocketRequester.Builder` for such cases with the expected codecs and applies any `RSocketConnectorConfigurer` bean.

The `RSocketRequester.Builder` instance is a prototype bean, meaning each injection point will provide you with a new instance . This is done on purpose since this builder is stateful and you should not create requesters with different setups using the same instance.

The following code shows a typical example:

Java

```
import reactor.core.publisher.Mono;

import org.springframework.messaging.rsocket.RSocketRequester;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    private final RSocketRequester rsocketRequester;

    public MyService(RSocketRequester.Builder rsocketRequesterBuilder) {
        this.rsocketRequester = rsocketRequesterBuilder.tcp("example.org", 9898);
    }

    public Mono<User> someRSocketCall(String name) {
        return
this.rsocketRequester.route("user").data(name).retrieveMono(User.class);
    }

}
```

Kotlin

```
import org.springframework.messaging.rsocket.RSocketRequester
import org.springframework.stereotype.Service
import reactor.core.publisher.Mono

@Service
class MyService(rsocketRequesterBuilder: RSocketRequester.Builder) {

    private val rsocketRequester: RSocketRequester

    init {
        rsocketRequester = rsocketRequesterBuilder.tcp("example.org", 9898)
    }

    fun someRSocketCall(name: String): Mono<User> {
        return rsocketRequester.route("user").data(name).retrieveMono(
            User::class.java
        )
    }

}
```

10.5. Spring Integration

Spring Boot offers several conveniences for working with Spring Integration, including the `spring-boot-starter-integration` "Starter". Spring Integration provides abstractions over messaging and also other transports such as HTTP, TCP, and others. If Spring Integration is available on your classpath, it is initialized through the `@EnableIntegration` annotation.

Spring Integration polling logic relies on the auto-configured `TaskScheduler`. The default `PollerMetadata` (poll unbounded number of messages every second) can be customized with `spring.integration.poller.*` configuration properties.

Spring Boot also configures some features that are triggered by the presence of additional Spring Integration modules. If `spring-integration-jmx` is also on the classpath, message processing statistics are published over JMX. If `spring-integration-jdbc` is available, the default database schema can be created on startup, as shown in the following line:

Properties

```
spring.integration.jdbc.initialize-schema=always
```

Yaml

```
spring:
  integration:
    jdbc:
      initialize-schema: "always"
```

If `spring-integration-rsocket` is available, developers can configure an RSocket server using `"spring.rsocket.server.*"` properties and let it use `IntegrationRSocketEndpoint` or `RSocketOutboundGateway` components to handle incoming RSocket messages. This infrastructure can handle Spring Integration RSocket channel adapters and `@MessageMapping` handlers (given `"spring.integration.rsocket.server.message-mapping-enabled"` is configured).

Spring Boot can also auto-configure an `ClientRSocketConnector` using configuration properties:

Properties

```
# Connecting to a RSocket server over TCP
spring.integration.rsocket.client.host=example.org
spring.integration.rsocket.client.port=9898
```

Yaml

```
# Connecting to a RSocket server over TCP
spring:
  integration:
    rsocket:
      client:
        host: "example.org"
        port: 9898
```

Properties

```
# Connecting to a RSocket Server over WebSocket
spring.integration.rsocket.client.uri=ws://example.org
```

Yaml

```
# Connecting to a RSocket Server over WebSocket
spring:
  integration:
    rsocket:
      client:
        uri: "ws://example.org"
```

See the `IntegrationAutoConfiguration` and `IntegrationProperties` classes for more details.

10.6. WebSockets

Spring Boot provides WebSockets auto-configuration for embedded Tomcat, Jetty, and Undertow. If you deploy a war file to a standalone container, Spring Boot assumes that the container is responsible for the configuration of its WebSocket support.

Spring Framework provides rich WebSocket support for MVC web applications that can be easily accessed through the `spring-boot-starter-websocket` module.

WebSocket support is also available for reactive web applications and requires to include the WebSocket API alongside `spring-boot-starter-webflux`:

```
<dependency>
    <groupId>jakarta.websocket</groupId>
    <artifactId>jakarta.websocket-api</artifactId>
</dependency>
```

10.7. What to Read Next

The next section describes how to enable IO capabilities in your application. You can read about caching, mail, validation, rest clients and more in this section.

Chapter 11. IO

Most applications will need to deal with input and output concerns at some point. Spring Boot provides utilities and integrations with a range of technologies to help when you need IO capabilities. This section covers standard IO features such as caching and validation as well as more advanced topics such as scheduling and distributed transactions. We will also cover calling remote REST or SOAP services and sending email.

11.1. Caching

The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to methods, thus reducing the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker. Spring Boot auto-configures the cache infrastructure as long as caching support is enabled by using the `@EnableCaching` annotation.

NOTE	Check the relevant section of the Spring Framework reference for more details.

In a nutshell, to add caching to an operation of your service add the relevant annotation to its method, as shown in the following example:

Java

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Component;

@Component
public class MyMathService {

    @Cacheable("piDecimals")
    public int computePiDecimal(int precision) {
        ...
    }

}
```

Kotlin

```
import org.springframework.cache.annotation.Cacheable
import org.springframework.stereotype.Component

@Component
class MyMathService {

    @Cacheable("piDecimals")
    fun computePiDecimal(precision: Int): Int {
        ...
    }

}
```

This example demonstrates the use of caching on a potentially costly operation. Before invoking `computePiDecimal`, the abstraction looks for an entry in the `piDecimals` cache that matches the `i` argument. If an entry is found, the content in the cache is immediately returned to the caller, and the method is not invoked. Otherwise, the method is invoked, and the cache is updated before returning the value.

CAUTION	You can also use the standard JSR-107 (JCache) annotations (such as `@CacheResult`) transparently. However, we strongly advise you to not mix and match the Spring Cache and JCache annotations.

If you do not add any specific cache library, Spring Boot auto-configures a simple provider that uses concurrent maps in memory. When a cache is required (such as `piDecimals` in the preceding example), this provider creates it for you. The simple provider is not really recommended for production usage, but it is great for getting started and making sure that you understand the features. When you have made up your mind about the cache provider to use, please make sure to read its documentation to figure out how to configure the caches that your application uses. Nearly all providers require you to explicitly configure every cache that you use in the application. Some offer a way to customize the default caches defined by the `spring.cache.cache-names` property.

TIP	It is also possible to transparently update or evict data from the cache.

11.1.1. Supported Cache Providers

The cache abstraction does not provide an actual store and relies on abstraction materialized by the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces.

If you have not defined a bean of type `CacheManager` or a `CacheResolver` named `cacheResolver` (see `CachingConfigurer`), Spring Boot tries to detect the following providers (in the indicated order):

1. Generic
2. JCache (JSR-107) (EhCache 3, Hazelcast, Infinispan, and others)
3. Hazelcast

4. Infinispan
5. Couchbase
6. Redis
7. Caffeine
8. Cache2k
9. Simple

Additionally, Spring Boot for Apache Geode provides auto-configuration for using Apache Geode as a cache provider.

TIP	If the `CacheManager` is auto-configured by Spring Boot, it is possible to *force* a particular cache provider by setting the `spring.cache.type` property. Use this property if you need to use no-op caches in certain environments (such as tests).
TIP	Use the `spring-boot-starter-cache` "Starter" to quickly add basic caching dependencies. The starter brings in `spring-context-support`. If you add dependencies manually, you must include `spring-context-support` in order to use the JCache or Caffeine support.

If the `CacheManager` is auto-configured by Spring Boot, you can further tune its configuration before it is fully initialized by exposing a bean that implements the `CacheManagerCustomizer` interface. The following example sets a flag to say that `null` values should not be passed down to the underlying map:

Java

```
import org.springframework.boot.autoconfigure.cache.CacheManagerCustomizer;
import org.springframework.cache.concurrent.ConcurrentMapCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyCacheManagerConfiguration {

    @Bean
    public CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer()
{
        return (cacheManager) -> cacheManager.setAllowNullValues(false);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.cache.CacheManagerCustomizer
import org.springframework.cache.concurrent.ConcurrentMapCacheManager
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyCacheManagerConfiguration {

    @Bean
    fun cacheManagerCustomizer(): CacheManagerCustomizer<ConcurrentMapCacheManager> {
        return CacheManagerCustomizer { cacheManager ->
            cacheManager.isAllowNullValues = false
        }
    }

}
```

NOTE	In the preceding example, an auto-configured `ConcurrentMapCacheManager` is expected. If that is not the case (either you provided your own config or a different cache provider was auto-configured), the customizer is not invoked at all. You can have as many customizers as you want, and you can also order them by using `@Order` or `Ordered`.

Generic

Generic caching is used if the context defines *at least* one `org.springframework.cache.Cache` bean. A `CacheManager` wrapping all beans of that type is created.

JCache (JSR-107)

JCache is bootstrapped through the presence of a `javax.cache.spi.CachingProvider` on the classpath (that is, a JSR-107 compliant caching library exists on the classpath), and the `JCacheCacheManager` is provided by the `spring-boot-starter-cache` "Starter". Various compliant libraries are available, and Spring Boot provides dependency management for Ehcache 3, Hazelcast, and Infinispan. Any other compliant library can be added as well.

It might happen that more than one provider is present, in which case the provider must be explicitly specified. Even if the JSR-107 standard does not enforce a standardized way to define the location of the configuration file, Spring Boot does its best to accommodate setting a cache with implementation details, as shown in the following example:

Properties

```
# Only necessary if more than one provider is present
spring.cache.jcache.provider=com.example.MyCachingProvider
spring.cache.jcache.config=classpath:example.xml
```

Yaml

```
# Only necessary if more than one provider is present
spring:
  cache:
    jcache:
      provider: "com.example.MyCachingProvider"
      config: "classpath:example.xml"
```

NOTE	When a cache library offers both a native implementation and JSR-107 support, Spring Boot prefers the JSR-107 support, so that the same features are available if you switch to a different JSR-107 implementation.

TIP	Spring Boot has general support for Hazelcast. If a single `HazelcastInstance` is available, it is automatically reused for the `CacheManager` as well, unless the `spring.cache.jcache.config` property is specified.

There are two ways to customize the underlying `javax.cache.cacheManager`:

- Caches can be created on startup by setting the `spring.cache.cache-names` property. If a custom `javax.cache.configuration.Configuration` bean is defined, it is used to customize them.
- `org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer` beans are invoked with the reference of the `CacheManager` for full customization.

TIP	If a standard `javax.cache.CacheManager` bean is defined, it is wrapped automatically in an `org.springframework.cache.CacheManager` implementation that the abstraction expects. No further customization is applied to it.

Hazelcast

Spring Boot has general support for Hazelcast. If a `HazelcastInstance` has been auto-configured and `com.hazelcast:hazelcast-spring` is on the classpath, it is automatically wrapped in a `CacheManager`.

NOTE	Hazelcast can be used as a JCache compliant cache or as a Spring `CacheManager` compliant cache. When setting `spring.cache.type` to `hazelcast`, Spring Boot will use the `CacheManager` based implementation. If you want to use Hazelcast as a JCache compliant cache, set `spring.cache.type` to `jcache`. If you have multiple JCache compliant cache providers and want to force the use of Hazelcast, you have to explicitly set the JCache provider.

Infinispan

Infinispan has no default configuration file location, so it must be specified explicitly. Otherwise, the default bootstrap is used.

Properties

```
spring.cache.infinispan.config=infinispan.xml
```

Yaml

```
spring:
  cache:
    infinispan:
      config: "infinispan.xml"
```

Caches can be created on startup by setting the `spring.cache.cache-names` property. If a custom `ConfigurationBuilder` bean is defined, it is used to customize the caches.

To be compatible with Spring Boot's Jakarta EE 9 baseline, Infinispan's `-jakarta` modules must be used. For every module with a `-jakarta` variant, the variant must be used in place of the standard module. For example, `infinispan-core-jakarta` and `infinispan-commons-jakarta` must be used in place of `infinispan-core` and `infinispan-commons` respectively.

Couchbase

If Spring Data Couchbase is available and Couchbase is configured, a `CouchbaseCacheManager` is auto-configured. It is possible to create additional caches on startup by setting the `spring.cache.cache-names` property and cache defaults can be configured by using `spring.cache.couchbase.*` properties. For instance, the following configuration creates `cache1` and `cache2` caches with an entry *expiration* of 10 minutes:

Properties

```
spring.cache.cache-names=cache1,cache2
spring.cache.couchbase.expiration=10m
```

Yaml

```
spring:
  cache:
    cache-names: "cache1,cache2"
    couchbase:
      expiration: "10m"
```

If you need more control over the configuration, consider registering a `CouchbaseCacheManagerBuilderCustomizer` bean. The following example shows a customizer that configures a specific entry expiration for `cache1` and `cache2`:

Java

```
import java.time.Duration;

import
org.springframework.boot.autoconfigure.cache.CouchbaseCacheManagerBuilderCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.couchbase.cache.CouchbaseCacheConfiguration;

@Configuration(proxyBeanMethods = false)
public class MyCouchbaseCacheManagerConfiguration {

    @Bean
    public CouchbaseCacheManagerBuilderCustomizer
myCouchbaseCacheManagerBuilderCustomizer() {
        return (builder) -> builder
                .withCacheConfiguration("cache1", CouchbaseCacheConfiguration
                        .defaultCacheConfig().entryExpiry(Duration.ofSeconds(10)))
                .withCacheConfiguration("cache2", CouchbaseCacheConfiguration
                        .defaultCacheConfig().entryExpiry(Duration.ofMinutes(1)));

    }

}
```

Kotlin

```
import
org.springframework.boot.autoconfigure.cache.CouchbaseCacheManagerBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.data.couchbase.cache.CouchbaseCacheConfiguration
import java.time.Duration

@Configuration(proxyBeanMethods = false)
class MyCouchbaseCacheManagerConfiguration {

    @Bean
    fun myCouchbaseCacheManagerBuilderCustomizer():
CouchbaseCacheManagerBuilderCustomizer {
        return CouchbaseCacheManagerBuilderCustomizer { builder ->
            builder
                .withCacheConfiguration(
                    "cache1", CouchbaseCacheConfiguration
                        .defaultCacheConfig().entryExpiry(Duration.ofSeconds(10))
                )
                .withCacheConfiguration(
                    "cache2", CouchbaseCacheConfiguration
                        .defaultCacheConfig().entryExpiry(Duration.ofMinutes(1))
                )
        }
    }

}
```

Redis

If Redis is available and configured, a `RedisCacheManager` is auto-configured. It is possible to create additional caches on startup by setting the `spring.cache.cache-names` property and cache defaults can be configured by using `spring.cache.redis.*` properties. For instance, the following configuration creates `cache1` and `cache2` caches with a *time to live* of 10 minutes:

Properties

```
spring.cache.cache-names=cache1,cache2
spring.cache.redis.time-to-live=10m
```

Yaml

```
spring:
  cache:
    cache-names: "cache1,cache2"
    redis:
      time-to-live: "10m"
```

NOTE	By default, a key prefix is added so that, if two separate caches use the same key, Redis does not have overlapping keys and cannot return invalid values. We strongly recommend keeping this setting enabled if you create your own `RedisCacheManager`.

TIP	You can take full control of the default configuration by adding a `RedisCacheConfiguration` `@Bean` of your own. This can be useful if you need to customize the default serialization strategy.

If you need more control over the configuration, consider registering a `RedisCacheManagerBuilderCustomizer` bean. The following example shows a customizer that configures a specific time to live for `cache1` and `cache2`:

Java

```
import java.time.Duration;

import
org.springframework.boot.autoconfigure.cache.RedisCacheManagerBuilderCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.cache.RedisCacheConfiguration;

@Configuration(proxyBeanMethods = false)
public class MyRedisCacheManagerConfiguration {

    @Bean
    public RedisCacheManagerBuilderCustomizer myRedisCacheManagerBuilderCustomizer() {
        return (builder) -> builder
                .withCacheConfiguration("cache1", RedisCacheConfiguration
                        .defaultCacheConfig().entryTtl(Duration.ofSeconds(10)))
                .withCacheConfiguration("cache2", RedisCacheConfiguration
                        .defaultCacheConfig().entryTtl(Duration.ofMinutes(1)));

    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.cache.RedisCacheManagerBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.data.redis.cache.RedisCacheConfiguration
import java.time.Duration

@Configuration(proxyBeanMethods = false)
class MyRedisCacheManagerConfiguration {

    @Bean
    fun myRedisCacheManagerBuilderCustomizer(): RedisCacheManagerBuilderCustomizer {
        return RedisCacheManagerBuilderCustomizer { builder ->
            builder
                .withCacheConfiguration(
                    "cache1", RedisCacheConfiguration
                        .defaultCacheConfig().entryTtl(Duration.ofSeconds(10))
                )
                .withCacheConfiguration(
                    "cache2", RedisCacheConfiguration
                        .defaultCacheConfig().entryTtl(Duration.ofMinutes(1))
                )
        }
    }

}
```

Caffeine

Caffeine is a Java 8 rewrite of Guava's cache that supersedes support for Guava. If Caffeine is present, a `CaffeineCacheManager` (provided by the `spring-boot-starter-cache` "Starter") is auto-configured. Caches can be created on startup by setting the `spring.cache.cache-names` property and can be customized by one of the following (in the indicated order):

1. A cache spec defined by `spring.cache.caffeine.spec`
2. A `com.github.benmanes.caffeine.cache.CaffeineSpec` bean is defined
3. A `com.github.benmanes.caffeine.cache.Caffeine` bean is defined

For instance, the following configuration creates `cache1` and `cache2` caches with a maximum size of 500 and a *time to live* of 10 minutes

Properties

```
spring.cache.cache-names=cache1,cache2
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s
```

Yaml

```
spring:
  cache:
    cache-names: "cache1,cache2"
    caffeine:
      spec: "maximumSize=500,expireAfterAccess=600s"
```

If a `com.github.benmanes.caffeine.cache.CacheLoader` bean is defined, it is automatically associated to the `CaffeineCacheManager`. Since the `CacheLoader` is going to be associated with *all* caches managed by the cache manager, it must be defined as `CacheLoader<Object, Object>`. The auto-configuration ignores any other generic type.

Cache2k

Cache2k is an in-memory cache. If the Cache2k spring integration is present, a `SpringCache2kCacheManager` is auto-configured.

Caches can be created on startup by setting the `spring.cache.cache-names` property. Cache defaults can be customized using a `Cache2kBuilderCustomizer` bean. The following example shows a customizer that configures the capacity of the cache to 200 entries, with an expiration of 5 minutes:

Java

```
import java.util.concurrent.TimeUnit;

import org.springframework.boot.autoconfigure.cache.Cache2kBuilderCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyCache2kDefaultsConfiguration {

    @Bean
    public Cache2kBuilderCustomizer myCache2kDefaultsCustomizer() {
        return (builder) -> builder.entryCapacity(200)
                .expireAfterWrite(5, TimeUnit.MINUTES);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.cache.Cache2kBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import java.util.concurrent.TimeUnit

@Configuration(proxyBeanMethods = false)
class MyCache2kDefaultsConfiguration {

    @Bean
    fun myCache2kDefaultsCustomizer(): Cache2kBuilderCustomizer {
        return Cache2kBuilderCustomizer { builder ->
            builder.entryCapacity(200)
                .expireAfterWrite(5, TimeUnit.MINUTES)
        }
    }
}
```

Simple

If none of the other providers can be found, a simple implementation using a `ConcurrentHashMap` as the cache store is configured. This is the default if no caching library is present in your application. By default, caches are created as needed, but you can restrict the list of available caches by setting the `cache-names` property. For instance, if you want only `cache1` and `cache2` caches, set the `cache-names` property as follows:

Properties

```
spring.cache.cache-names=cache1,cache2
```

Yaml

```
spring:
  cache:
    cache-names: "cache1,cache2"
```

If you do so and your application uses a cache not listed, then it fails at runtime when the cache is needed, but not on startup. This is similar to the way the "real" cache providers behave if you use an undeclared cache.

None

When `@EnableCaching` is present in your configuration, a suitable cache configuration is expected as well. If you have a custom `CacheManager`, consider defining it in a separate `@Configuration` class so that you can override it if necessary. None uses a no-op implementation that is useful in tests, and slice tests use that by default via `@AutoConfigureCache`.

If you need to use a no-op cache rather than the auto-configured cache manager in a certain

environment, set the cache type to `none`, as shown in the following example:

Properties

```
spring.cache.type=none
```

Yaml

```
spring:
  cache:
    type: "none"
```

11.2. Hazelcast

If Hazelcast is on the classpath and a suitable configuration is found, Spring Boot auto-configures a `HazelcastInstance` that you can inject in your application.

Spring Boot first attempts to create a client by checking the following configuration options:

- The presence of a `com.hazelcast.client.config.ClientConfig` bean.
- A configuration file defined by the `spring.hazelcast.config` property.
- The presence of the `hazelcast.client.config` system property.
- A `hazelcast-client.xml` in the working directory or at the root of the classpath.
- A `hazelcast-client.yaml` (or `hazelcast-client.yml`) in the working directory or at the root of the classpath.

If a client can not be created, Spring Boot attempts to configure an embedded server. If you define a `com.hazelcast.config.Config` bean, Spring Boot uses that. If your configuration defines an instance name, Spring Boot tries to locate an existing instance rather than creating a new one.

You could also specify the Hazelcast configuration file to use through configuration, as shown in the following example:

Properties

```
spring.hazelcast.config=classpath:config/my-hazelcast.xml
```

Yaml

```
spring:
  hazelcast:
    config: "classpath:config/my-hazelcast.xml"
```

Otherwise, Spring Boot tries to find the Hazelcast configuration from the default locations: `hazelcast.xml` in the working directory or at the root of the classpath, or a YAML counterpart in the same locations. We also check if the `hazelcast.config` system property is set. See the Hazelcast

documentation for more details.

TIP	By default, `@SpringAware` on Hazelcast components is supported. The `ManagementContext` can be overridden by declaring a `HazelcastConfigCustomizer` bean with an `@Order` higher than zero.

NOTE	Spring Boot also has explicit caching support for Hazelcast. If caching is enabled, the `HazelcastInstance` is automatically wrapped in a `CacheManager` implementation.

11.3. Quartz Scheduler

Spring Boot offers several conveniences for working with the Quartz scheduler, including the `spring-boot-starter-quartz` "Starter". If Quartz is available, a `Scheduler` is auto-configured (through the `SchedulerFactoryBean` abstraction).

Beans of the following types are automatically picked up and associated with the `Scheduler`:

- `JobDetail`: defines a particular Job. `JobDetail` instances can be built with the `JobBuilder` API.
- `Calendar`.
- `Trigger`: defines when a particular job is triggered.

By default, an in-memory `JobStore` is used. However, it is possible to configure a JDBC-based store if a `DataSource` bean is available in your application and if the `spring.quartz.job-store-type` property is configured accordingly, as shown in the following example:

Properties

```
spring.quartz.job-store-type=jdbc
```

Yaml

```
spring:
  quartz:
    job-store-type: "jdbc"
```

When the JDBC store is used, the schema can be initialized on startup, as shown in the following example:

Properties

```
spring.quartz.jdbc.initialize-schema=always
```

Yaml

```
spring:
  quartz:
    jdbc:
      initialize-schema: "always"
```

WARNING	By default, the database is detected and initialized by using the standard scripts provided with the Quartz library. These scripts drop existing tables, deleting all triggers on every restart. It is also possible to provide a custom script by setting the `spring.quartz.jdbc.schema` property.

To have Quartz use a `DataSource` other than the application's main `DataSource`, declare a `DataSource` bean, annotating its `@Bean` method with `@QuartzDataSource`. Doing so ensures that the Quartz-specific `DataSource` is used by both the `SchedulerFactoryBean` and for schema initialization. Similarly, to have Quartz use a `TransactionManager` other than the application's main `TransactionManager` declare a `TransactionManager` bean, annotating its `@Bean` method with `@QuartzTransactionManager`.

By default, jobs created by configuration will not overwrite already registered jobs that have been read from a persistent job store. To enable overwriting existing job definitions set the `spring.quartz.overwrite-existing-jobs` property.

Quartz Scheduler configuration can be customized using `spring.quartz` properties and `SchedulerFactoryBeanCustomizer` beans, which allow programmatic `SchedulerFactoryBean` customization. Advanced Quartz configuration properties can be customized using `spring.quartz.properties.*`.

NOTE	In particular, an `Executor` bean is not associated with the scheduler as Quartz offers a way to configure the scheduler through `spring.quartz.properties`. If you need to customize the task executor, consider implementing `SchedulerFactoryBeanCustomizer`.

Jobs can define setters to inject data map properties. Regular beans can also be injected in a similar manner, as shown in the following example:

Java

```
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

import org.springframework.scheduling.quartz.QuartzJobBean;

public class MySampleJob extends QuartzJobBean {

    private MyService myService;

    private String name;


    // Inject "MyService" bean
    public void setMyService(MyService myService) {
        this.myService = myService;
    }

    // Inject the "name" job data property
    public void setName(String name) {
        this.name = name;
    }

    @Override
    protected void executeInternal(JobExecutionContext context) throws
JobExecutionException {
        this.myService.someMethod(context.getFireTime(), this.name);
    }

}
```

Kotlin

```
import org.quartz.JobExecutionContext
import org.springframework.scheduling.quartz.QuartzJobBean

class MySampleJob : QuartzJobBean() {

    private var myService: MyService? = null

    private var name: String? = null

    // Inject "MyService" bean
    fun setMyService(myService: MyService?) {
        this.myService = myService
    }

    // Inject the "name" job data property
    fun setName(name: String?) {
        this.name = name
    }

    override fun executeInternal(context: JobExecutionContext) {
        myService!!.someMethod(context.fireTime, name)
    }

}
```

11.4. Sending Email

The Spring Framework provides an abstraction for sending email by using the `JavaMailSender` interface, and Spring Boot provides auto-configuration for it as well as a starter module.

TIP See the reference documentation for a detailed explanation of how you can use `JavaMailSender`.

If `spring.mail.host` and the relevant libraries (as defined by `spring-boot-starter-mail`) are available, a default `JavaMailSender` is created if none exists. The sender can be further customized by configuration items from the `spring.mail` namespace. See `MailProperties` for more details.

In particular, certain default timeout values are infinite, and you may want to change that to avoid having a thread blocked by an unresponsive mail server, as shown in the following example:

Properties

```
spring.mail.properties[mail.smtp.connectiontimeout]=5000
spring.mail.properties[mail.smtp.timeout]=3000
spring.mail.properties[mail.smtp.writetimeout]=5000
```

Yaml

```
spring:
  mail:
    properties:
      "[mail.smtp.connectiontimeout]": 5000
      "[mail.smtp.timeout]": 3000
      "[mail.smtp.writetimeout]": 5000
```

It is also possible to configure a `JavaMailSender` with an existing `Session` from JNDI:

Properties

```
spring.mail.jndi-name=mail/Session
```

Yaml

```
spring:
  mail:
    jndi-name: "mail/Session"
```

When a `jndi-name` is set, it takes precedence over all other Session-related settings.

11.5. Validation

The method validation feature supported by Bean Validation 1.1 is automatically enabled as long as a JSR-303 implementation (such as Hibernate validator) is on the classpath. This lets bean methods be annotated with `jakarta.validation` constraints on their parameters and/or on their return value. Target classes with such annotated methods need to be annotated with the `@Validated` annotation at the type level for their methods to be searched for inline constraint annotations.

For instance, the following service triggers the validation of the first argument, making sure its size is between 8 and 10:

Java

```
import jakarta.validation.constraints.Size;

import org.springframework.stereotype.Service;
import org.springframework.validation.annotation.Validated;

@Service
@Validated
public class MyBean {

    public Archive findByCodeAndAuthor(@Size(min = 8, max = 10) String code, Author
author) {
        return ...
    }

}
```

Kotlin

```
import jakarta.validation.constraints.Size
import org.springframework.stereotype.Service
import org.springframework.validation.annotation.Validated

@Service
@Validated
class MyBean {

    fun findByCodeAndAuthor(code: @Size(min = 8, max = 10) String?, author: Author?):
Archive? {
        return null
    }

}
```

The application's `MessageSource` is used when resolving `{parameters}` in constraint messages. This allows you to use your application's `messages.properties` files for Bean Validation messages. Once the parameters have been resolved, message interpolation is completed using Bean Validation's default interpolator.

To customize the `Configuration` used to build the `ValidatorFactory`, define a `ValidationConfigurationCustomizer` bean. When multiple customizer beans are defined, they are called in order based on their `@Order` annotation or `Ordered` implementation.

11.6. Calling REST Services

If your application calls remote REST services, Spring Boot makes that very convenient using a `RestTemplate` or a `WebClient`.

11.6.1. RestTemplate

If you need to call remote REST services from your application, you can use the Spring Framework's `RestTemplate` class. Since `RestTemplate` instances often need to be customized before being used, Spring Boot does not provide any single auto-configured `RestTemplate` bean. It does, however, auto-configure a `RestTemplateBuilder`, which can be used to create `RestTemplate` instances when needed. The auto-configured `RestTemplateBuilder` ensures that sensible `HttpMessageConverters` are applied to `RestTemplate` instances.

The following code shows a typical example:

Java

```
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyService {

    private final RestTemplate restTemplate;

    public MyService(RestTemplateBuilder restTemplateBuilder) {
        this.restTemplate = restTemplateBuilder.build();
    }

    public Details someRestCall(String name) {
        return this.restTemplate.getForObject("/{name}/details", Details.class, name);
    }

}
```

Kotlin

```
import org.springframework.boot.web.client.RestTemplateBuilder
import org.springframework.stereotype.Service
import org.springframework.web.client.RestTemplate

@Service
class MyService(restTemplateBuilder: RestTemplateBuilder) {

    private val restTemplate: RestTemplate

    init {
        restTemplate = restTemplateBuilder.build()
    }

    fun someRestCall(name: String): Details {
        return restTemplate.getForObject("/{name}/details", Details::class.java,
name)!!
    }

}
```

`RestTemplateBuilder` includes a number of useful methods that can be used to quickly configure a `RestTemplate`. For example, to add BASIC authentication support, you can use `builder.basicAuthentication("user", "password").build()`.

RestTemplate HTTP Client

Spring Boot will auto-detect which HTTP client to use with `RestTemplate` depending on the libraries available on the application classpath. In order of preference, the following clients are supported:

1. Apache HttpClient
2. OkHttp
3. Simple JDK client (`HttpURLConnection`)

If multiple clients are available on the classpath, the most preferred client will be used.

RestTemplate Customization

There are three main approaches to `RestTemplate` customization, depending on how broadly you want the customizations to apply.

To make the scope of any customizations as narrow as possible, inject the auto-configured `RestTemplateBuilder` and then call its methods as required. Each method call returns a new `RestTemplateBuilder` instance, so the customizations only affect this use of the builder.

To make an application-wide, additive customization, use a `RestTemplateCustomizer` bean. All such beans are automatically registered with the auto-configured `RestTemplateBuilder` and are applied to any templates that are built with it.

The following example shows a customizer that configures the use of a proxy for all hosts except `192.168.0.5`:

Java

```
import org.apache.hc.client5.http.classic.HttpClient;
import org.apache.hc.client5.http.impl.classic.HttpClientBuilder;
import org.apache.hc.client5.http.impl.routing.DefaultProxyRoutePlanner;
import org.apache.hc.client5.http.routing.HttpRoutePlanner;
import org.apache.hc.core5.http.HttpException;
import org.apache.hc.core5.http.HttpHost;
import org.apache.hc.core5.http.protocol.HttpContext;

import org.springframework.boot.web.client.RestTemplateCustomizer;
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
import org.springframework.web.client.RestTemplate;

public class MyRestTemplateCustomizer implements RestTemplateCustomizer {

    @Override
    public void customize(RestTemplate restTemplate) {
        HttpRoutePlanner routePlanner = new CustomRoutePlanner(new
HttpHost("proxy.example.com"));
        HttpClient httpClient =
HttpClientBuilder.create().setRoutePlanner(routePlanner).build();
        restTemplate.setRequestFactory(new
HttpComponentsClientHttpRequestFactory(httpClient));
    }

    static class CustomRoutePlanner extends DefaultProxyRoutePlanner {

        CustomRoutePlanner(HttpHost proxy) {
            super(proxy);
        }

        @Override
        protected HttpHost determineProxy(HttpHost target, HttpContext context) throws
HttpException {
            if (target.getHostName().equals("192.168.0.5")) {
                return null;
            }
            return super.determineProxy(target, context);
        }

    }

}
```

Kotlin

```
import org.apache.hc.client5.http.classic.HttpClient
import org.apache.hc.client5.http.impl.classic.HttpClientBuilder
import org.apache.hc.client5.http.impl.routing.DefaultProxyRoutePlanner
import org.apache.hc.client5.http.routing.HttpRoutePlanner
import org.apache.hc.core5.http.HttpException
import org.apache.hc.core5.http.HttpHost
import org.apache.hc.core5.http.protocol.HttpContext
import org.springframework.boot.web.client.RestTemplateCustomizer
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory
import org.springframework.web.client.RestTemplate

class MyRestTemplateCustomizer : RestTemplateCustomizer {

    override fun customize(restTemplate: RestTemplate) {
        val routePlanner: HttpRoutePlanner =
CustomRoutePlanner(HttpHost("proxy.example.com"))
        val httpClient: HttpClient =
HttpClientBuilder.create().setRoutePlanner(routePlanner).build()
        restTemplate.requestFactory =
HttpComponentsClientHttpRequestFactory(httpClient)
    }

    internal class CustomRoutePlanner(proxy: HttpHost?) :
DefaultProxyRoutePlanner(proxy) {

        @Throws(HttpException::class)
        public override fun determineProxy(target: HttpHost, context: HttpContext):
HttpHost? {
            if (target.hostName == "192.168.0.5") {
                return null
            }
            return  super.determineProxy(target, context)
        }

    }

}
```

Finally, you can define your own `RestTemplateBuilder` bean. Doing so will replace the auto-configured builder. If you want any `RestTemplateCustomizer` beans to be applied to your custom builder, as the auto-configuration would have done, configure it using a `RestTemplateBuilderConfigurer`. The following example exposes a `RestTemplateBuilder` that matches what Spring Boot's auto-configuration would have done, except that custom connect and read timeouts are also specified:

Java

```
import java.time.Duration;

import
org.springframework.boot.autoconfigure.web.client.RestTemplateBuilderConfigurer;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyRestTemplateBuilderConfiguration {

    @Bean
    public RestTemplateBuilder restTemplateBuilder(RestTemplateBuilderConfigurer
configurer) {
        return configurer.configure(new RestTemplateBuilder())
            .setConnectTimeout(Duration.ofSeconds(5))
            .setReadTimeout(Duration.ofSeconds(2));
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.web.client.RestTemplateBuilderConfigurer
import org.springframework.boot.web.client.RestTemplateBuilder
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import java.time.Duration

@Configuration(proxyBeanMethods = false)
class MyRestTemplateBuilderConfiguration {

    @Bean
    fun restTemplateBuilder(configurer: RestTemplateBuilderConfigurer):
RestTemplateBuilder {
        return
configurer.configure(RestTemplateBuilder()).setConnectTimeout(Duration.ofSeconds(5))
            .setReadTimeout(Duration.ofSeconds(2))
    }

}
```

The most extreme (and rarely used) option is to create your own `RestTemplateBuilder` bean without using a configurer. In addition to replacing the auto-configured builder, this also prevents any `RestTemplateCustomizer` beans from being used.

RestTemplate SSL Support

If you need custom SSL configuration on the `RestTemplate`, you can apply an SSL bundle to the `RestTemplateBuilder` as shown in this example:

Java

```
import org.springframework.boot.docs.io.restclient.resttemplate.Details;
import org.springframework.boot.ssl.SslBundles;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyService {

    private final RestTemplate restTemplate;

    public MyService(RestTemplateBuilder restTemplateBuilder, SslBundles sslBundles) {
        this.restTemplate =
restTemplateBuilder.setSslBundle(sslBundles.getBundle("mybundle")).build();
    }

    public Details someRestCall(String name) {
        return this.restTemplate.getForObject("/{name}/details", Details.class, name);
    }

}
```

Kotlin

```
import org.springframework.boot.docs.io.restclient.resttemplate.Details
import org.springframework.boot.ssl.SslBundles
import org.springframework.boot.web.client.RestTemplateBuilder
import org.springframework.stereotype.Service
import org.springframework.web.client.RestTemplate

@Service
class MyService(restTemplateBuilder: RestTemplateBuilder, sslBundles: SslBundles) {

    private val restTemplate: RestTemplate

    init {
        restTemplate =
restTemplateBuilder.setSslBundle(sslBundles.getBundle("mybundle")).build()
    }

    fun someRestCall(name: String): Details {
        return restTemplate.getForObject("/{name}/details", Details::class.java,
name)!!
    }

}
```

11.6.2. WebClient

If you have Spring WebFlux on your classpath, you can also choose to use `WebClient` to call remote REST services. Compared to `RestTemplate`, this client has a more functional feel and is fully reactive. You can learn more about the `WebClient` in the dedicated section in the Spring Framework docs.

Spring Boot creates and pre-configures a `WebClient.Builder` for you. It is strongly advised to inject it in your components and use it to create `WebClient` instances. Spring Boot is configuring that builder to share HTTP resources, reflect codecs setup in the same fashion as the server ones (see WebFlux HTTP codecs auto-configuration), and more.

The following code shows a typical example:

Java

```
import reactor.core.publisher.Mono;

import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;

@Service
public class MyService {

    private final WebClient webClient;

    public MyService(WebClient.Builder webClientBuilder) {
        this.webClient = webClientBuilder.baseUrl("https://example.org").build();
    }

    public Mono<Details> someRestCall(String name) {
        return this.webClient.get().uri("/{name}/details",
name).retrieve().bodyToMono(Details.class);
    }

}
```

Kotlin

```
import org.springframework.stereotype.Service
import org.springframework.web.reactive.function.client.WebClient
import reactor.core.publisher.Mono

@Service
class MyService(webClientBuilder: WebClient.Builder) {

    private val webClient: WebClient

    init {
        webClient = webClientBuilder.baseUrl("https://example.org").build()
    }

    fun someRestCall(name: String?): Mono<Details> {
        return webClient.get().uri("/{name}/details", name)
                .retrieve().bodyToMono(Details::class.java)
    }

}
```

WebClient Runtime

Spring Boot will auto-detect which `ClientHttpConnector` to use to drive `WebClient` depending on the libraries available on the application classpath. In order of preference, the following clients are supported:

1. Reactor Netty
2. Jetty RS client
3. Apache HttpClient
4. JDK HttpClient

If multiple clients are available on the classpath, the most preferred client will be used.

The `spring-boot-starter-webflux` starter depends on `io.projectreactor.netty:reactor-netty` by default, which brings both server and client implementations. If you choose to use Jetty as a reactive server instead, you should add a dependency on the Jetty Reactive HTTP client library, `org.eclipse.jetty:jetty-reactive-httpclient`. Using the same technology for server and client has its advantages, as it will automatically share HTTP resources between client and server.

Developers can override the resource configuration for Jetty and Reactor Netty by providing a custom `ReactorResourceFactory` or `JettyResourceFactory` bean - this will be applied to both clients and servers.

If you wish to override that choice for the client, you can define your own `ClientHttpConnector` bean and have full control over the client configuration.

You can learn more about the `WebClient` configuration options in the Spring Framework reference documentation.

WebClient Customization

There are three main approaches to `WebClient` customization, depending on how broadly you want the customizations to apply.

To make the scope of any customizations as narrow as possible, inject the auto-configured `WebClient.Builder` and then call its methods as required. `WebClient.Builder` instances are stateful: Any change on the builder is reflected in all clients subsequently created with it. If you want to create several clients with the same builder, you can also consider cloning the builder with `WebClient.Builder other = builder.clone();`.

To make an application-wide, additive customization to all `WebClient.Builder` instances, you can declare `WebClientCustomizer` beans and change the `WebClient.Builder` locally at the point of injection.

Finally, you can fall back to the original API and use `WebClient.create()`. In that case, no auto-configuration or `WebClientCustomizer` is applied.

WebClient SSL Support

If you need custom SSL configuration on the `ClientHttpConnector` used by the `WebClient`, you can inject a `WebClientSsl` instance that can be used with the builder's `apply` method.

The `WebClientSsl` interface provides access to any SSL bundles that you have defined in your `application.properties` or `application.yaml` file.

The following code shows a typical example:

Java

```
import reactor.core.publisher.Mono;

import
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientSsl;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;

@Service
public class MyService {

    private final WebClient webClient;

    public MyService(WebClient.Builder webClientBuilder, WebClientSsl ssl) {
        this.webClient =
webClientBuilder.baseUrl("https://example.org").apply(ssl.fromBundle("mybundle")).buil
d();
    }

    public Mono<Details> someRestCall(String name) {
        return this.webClient.get().uri("/{name}/details",
name).retrieve().bodyToMono(Details.class);
    }

}
```

Kotlin

```
import
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientSsl
import org.springframework.stereotype.Service
import org.springframework.web.reactive.function.client.WebClient
import reactor.core.publisher.Mono

@Service
class MyService(webClientBuilder: WebClient.Builder, ssl: WebClientSsl) {

    private val webClient: WebClient

    init {
        webClient = webClientBuilder.baseUrl("https://example.org")
                .apply(ssl.fromBundle("mybundle")).build()
    }

    fun someRestCall(name: String?): Mono<Details> {
        return webClient.get().uri("/{name}/details", name)
                .retrieve().bodyToMono(Details::class.java)
    }

}
```

11.7. Web Services

Spring Boot provides Web Services auto-configuration so that all you must do is define your `Endpoints`.

The Spring Web Services features can be easily accessed with the `spring-boot-starter-webservices` module.

`SimpleWsdl11Definition` and `SimpleXsdSchema` beans can be automatically created for your WSDLs and XSDs respectively. To do so, configure their location, as shown in the following example:

Properties

```
spring.webservices.wsdl-locations=classpath:/wsdl
```

Yaml

```
spring:
  webservices:
    wsdl-locations: "classpath:/wsdl"
```

11.7.1. Calling Web Services with WebServiceTemplate

If you need to call remote Web services from your application, you can use the `WebServiceTemplate` class. Since `WebServiceTemplate` instances often need to be customized before being used, Spring Boot does not provide any single auto-configured `WebServiceTemplate` bean. It does, however, auto-configure a `WebServiceTemplateBuilder`, which can be used to create `WebServiceTemplate` instances when needed.

The following code shows a typical example:

Java

```
import org.springframework.boot.webservices.client.WebServiceTemplateBuilder;
import org.springframework.stereotype.Service;
import org.springframework.ws.client.core.WebServiceTemplate;
import org.springframework.ws.soap.client.core.SoapActionCallback;

@Service
public class MyService {

    private final WebServiceTemplate webServiceTemplate;

    public MyService(WebServiceTemplateBuilder webServiceTemplateBuilder) {
        this.webServiceTemplate = webServiceTemplateBuilder.build();
    }

    public SomeResponse someWsCall(SomeRequest detailsReq) {
        return (SomeResponse)
this.webServiceTemplate.marshalSendAndReceive(detailsReq,
                new SoapActionCallback("https://ws.example.com/action"));
    }

}
```

Kotlin

```
import org.springframework.boot.webservices.client.WebServiceTemplateBuilder
import org.springframework.stereotype.Service
import org.springframework.ws.client.core.WebServiceTemplate
import org.springframework.ws.soap.client.core.SoapActionCallback

@Service
class MyService(webServiceTemplateBuilder: WebServiceTemplateBuilder) {

    private val webServiceTemplate: WebServiceTemplate

    init {
        webServiceTemplate = webServiceTemplateBuilder.build()
    }

    fun someWsCall(detailsReq: SomeRequest?): SomeResponse {
        return webServiceTemplate.marshalSendAndReceive(
            detailsReq,
            SoapActionCallback("https://ws.example.com/action")
        ) as SomeResponse
    }

}
```

By default, `WebServiceTemplateBuilder` detects a suitable HTTP-based `WebServiceMessageSender` using the available HTTP client libraries on the classpath. You can also customize read and connection timeouts as follows:

Java

```
import java.time.Duration;

import org.springframework.boot.webservices.client.HttpWebServiceMessageSenderBuilder;
import org.springframework.boot.webservices.client.WebServiceTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.ws.client.core.WebServiceTemplate;
import org.springframework.ws.transport.WebServiceMessageSender;

@Configuration(proxyBeanMethods = false)
public class MyWebServiceTemplateConfiguration {

    @Bean
    public WebServiceTemplate webServiceTemplate(WebServiceTemplateBuilder builder) {
        WebServiceMessageSender sender = new HttpWebServiceMessageSenderBuilder()
                .setConnectTimeout(Duration.ofSeconds(5))
                .setReadTimeout(Duration.ofSeconds(2))
                .build();
        return builder.messageSenders(sender).build();
    }

}
```

Kotlin

```
import org.springframework.boot.webservices.client.HttpWebServiceMessageSenderBuilder
import org.springframework.boot.webservices.client.WebServiceTemplateBuilder
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.ws.client.core.WebServiceTemplate
import java.time.Duration

@Configuration(proxyBeanMethods = false)
class MyWebServiceTemplateConfiguration {

    @Bean
    fun webServiceTemplate(builder: WebServiceTemplateBuilder): WebServiceTemplate {
        val sender = HttpWebServiceMessageSenderBuilder()
            .setConnectTimeout(Duration.ofSeconds(5))
            .setReadTimeout(Duration.ofSeconds(2))
            .build()
        return builder.messageSenders(sender).build()
    }

}
```

11.8. Distributed Transactions With JTA

Spring Boot supports distributed JTA transactions across multiple XA resources by using a transaction manager retrieved from JNDI.

When a JTA environment is detected, Spring's `JtaTransactionManager` is used to manage transactions. Auto-configured JMS, DataSource, and JPA beans are upgraded to support XA transactions. You can use standard Spring idioms, such as `@Transactional`, to participate in a distributed transaction. If you are within a JTA environment and still want to use local transactions, you can set the `spring.jta.enabled` property to `false` to disable the JTA auto-configuration.

11.8.1. Using a Jakarta EE Managed Transaction Manager

If you package your Spring Boot application as a `war` or `ear` file and deploy it to a Jakarta EE application server, you can use your application server's built-in transaction manager. Spring Boot tries to auto-configure a transaction manager by looking at common JNDI locations (`java:comp/UserTransaction`, `java:comp/TransactionManager`, and so on). When using a transaction service provided by your application server, you generally also want to ensure that all resources are managed by the server and exposed over JNDI. Spring Boot tries to auto-configure JMS by looking for a `ConnectionFactory` at the JNDI path (`java:/JmsXA` or `java:/XAConnectionFactory`), and you can use the `spring.datasource.jndi-name` property to configure your `DataSource`.

11.8.2. Mixing XA and Non-XA JMS Connections

When using JTA, the primary JMS `ConnectionFactory` bean is XA-aware and participates in distributed transactions. You can inject into your bean without needing to use any `@Qualifier`:

Java

```
public MyBean(ConnectionFactory connectionFactory) {
    // ...
}
```

Kotlin

```

```

In some situations, you might want to process certain JMS messages by using a non-XA `ConnectionFactory`. For example, your JMS processing logic might take longer than the XA timeout.

If you want to use a non-XA `ConnectionFactory`, you can the `nonXaJmsConnectionFactory` bean:

Java

```
public MyBean(@Qualifier("nonXaJmsConnectionFactory") ConnectionFactory
connectionFactory) {
    // ...
}
```

Kotlin

For consistency, the `jmsConnectionFactory` bean is also provided by using the bean alias `xaJmsConnectionFactory`:

Java

```
public MyBean(@Qualifier("xaJmsConnectionFactory") ConnectionFactory
connectionFactory) {
    // ...
}
```

Kotlin

11.8.3. Supporting an Embedded Transaction Manager

The `XAConnectionFactoryWrapper` and `XADataSourceWrapper` interfaces can be used to support embedded transaction managers. The interfaces are responsible for wrapping `XAConnectionFactory` and `XADataSource` beans and exposing them as regular `ConnectionFactory` and `DataSource` beans, which transparently enroll in the distributed transaction. DataSource and JMS auto-configuration use JTA variants, provided you have a `JtaTransactionManager` bean and appropriate XA wrapper beans registered within your `ApplicationContext`.

11.9. What to Read Next

You should now have a good understanding of Spring Boot's core features and the various technologies that Spring Boot provides support for through auto-configuration.

The next few sections go into detail about deploying applications to cloud platforms. You can read about building container images in the next section or skip to the production-ready features section.

Chapter 12. Container Images

Spring Boot applications can be containerized using Dockerfiles, or by using Cloud Native Buildpacks to create optimized docker compatible container images that you can run anywhere.

12.1. Efficient Container Images

It is easily possible to package a Spring Boot fat jar as a docker image. However, there are various downsides to copying and running the fat jar as is in the docker image. There's always a certain amount of overhead when running a fat jar without unpacking it, and in a containerized environment this can be noticeable. The other issue is that putting your application's code and all its dependencies in one layer in the Docker image is sub-optimal. Since you probably recompile your code more often than you upgrade the version of Spring Boot you use, it's often better to separate things a bit more. If you put jar files in the layer before your application classes, Docker often only needs to change the very bottom layer and can pick others up from its cache.

12.1.1. Layering Docker Images

To make it easier to create optimized Docker images, Spring Boot supports adding a layer index file to the jar. It provides a list of layers and the parts of the jar that should be contained within them. The list of layers in the index is ordered based on the order in which the layers should be added to the Docker/OCI image. Out-of-the-box, the following layers are supported:

- `dependencies` (for regular released dependencies)
- `spring-boot-loader` (for everything under `org/springframework/boot/loader`)
- `snapshot-dependencies` (for snapshot dependencies)
- `application` (for application classes and resources)

The following shows an example of a `layers.idx` file:

```
- "dependencies":
  - BOOT-INF/lib/library1.jar
  - BOOT-INF/lib/library2.jar
- "spring-boot-loader":
  - org/springframework/boot/loader/JarLauncher.class
  - org/springframework/boot/loader/jar/JarEntry.class
- "snapshot-dependencies":
  - BOOT-INF/lib/library3-SNAPSHOT.jar
- "application":
  - META-INF/MANIFEST.MF
  - BOOT-INF/classes/a/b/C.class
```

This layering is designed to separate code based on how likely it is to change between application builds. Library code is less likely to change between builds, so it is placed in its own layers to allow tooling to re-use the layers from cache. Application code is more likely to change between builds so it is isolated in a separate layer.

Spring Boot also supports layering for war files with the help of a `layers.idx`.

For Maven, see the packaging layered jar or war section for more details on adding a layer index to the archive. For Gradle, see the packaging layered jar or war section of the Gradle plugin documentation.

12.2. Dockerfiles

While it is possible to convert a Spring Boot fat jar into a docker image with just a few lines in the Dockerfile, we will use the layering feature to create an optimized docker image. When you create a jar containing the layers index file, the `spring-boot-jarmode-layertools` jar will be added as a dependency to your jar. With this jar on the classpath, you can launch your application in a special mode which allows the bootstrap code to run something entirely different from your application, for example, something that extracts the layers.

CAUTION	The `layertools` mode can not be used with a fully executable Spring Boot archive that includes a launch script. Disable launch script configuration when building a jar file that is intended to be used with `layertools`.

Here's how you can launch your jar with a `layertools` jar mode:

```
$ java -Djarmode=layertools -jar my-app.jar
```

This will provide the following output:

```
Usage:
  java -Djarmode=layertools -jar my-app.jar

Available commands:
  list     List layers from the jar that can be extracted
  extract  Extracts layers from the jar for image creation
  help     Help about any command
```

The `extract` command can be used to easily split the application into layers to be added to the dockerfile. Here is an example of a Dockerfile using `jarmode`.

```
FROM eclipse-temurin:17-jre as builder
WORKDIR application
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layertools -jar application.jar extract

FROM eclipse-temurin:17-jre
WORKDIR application
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/spring-boot-loader/ ./
COPY --from=builder application/snapshot-dependencies/ ./
COPY --from=builder application/application/ ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

Assuming the above `Dockerfile` is in the current directory, your docker image can be built with `docker build .`, or optionally specifying the path to your application jar, as shown in the following example:

```
$ docker build --build-arg JAR_FILE=path/to/myapp.jar .
```

This is a multi-stage dockerfile. The builder stage extracts the directories that are needed later. Each of the `COPY` commands relates to the layers extracted by the jarmode.

Of course, a Dockerfile can be written without using the jarmode. You can use some combination of `unzip` and `mv` to move things to the right layer but jarmode simplifies that.

12.3. Cloud Native Buildpacks

Dockerfiles are just one way to build docker images. Another way to build docker images is directly from your Maven or Gradle plugin, using buildpacks. If you've ever used an application platform such as Cloud Foundry or Heroku then you've probably used a buildpack. Buildpacks are the part of the platform that takes your application and converts it into something that the platform can actually run. For example, Cloud Foundry's Java buildpack will notice that you're pushing a `.jar` file and automatically add a relevant JRE.

With Cloud Native Buildpacks, you can create Docker compatible images that you can run anywhere. Spring Boot includes buildpack support directly for both Maven and Gradle. This means you can just type a single command and quickly get a sensible image into your locally running Docker daemon.

See the individual plugin documentation on how to use buildpacks with Maven and Gradle.

NOTE The Paketo Spring Boot buildpack supports the `layers.idx` file, so any customization that is applied to it will be reflected in the image created by the buildpack.

NOTE

In order to achieve reproducible builds and container image caching, Buildpacks can manipulate the application resources metadata (such as the file "last modified" information). You should ensure that your application does not rely on that metadata at runtime. Spring Boot can use that information when serving static resources, but this can be disabled with `spring.web.resources.cache.use-last-modified`.

12.4. What to Read Next

Once you've learned how to build efficient container images, you can read about deploying applications to a cloud platform, such as Kubernetes.

Chapter 13. Production-ready Features

Spring Boot includes a number of additional features to help you monitor and manage your application when you push it to production. You can choose to manage and monitor your application by using HTTP endpoints or with JMX. Auditing, health, and metrics gathering can also be automatically applied to your application.

13.1. Enabling Production-ready Features

The `spring-boot-actuator` module provides all of Spring Boot's production-ready features. The recommended way to enable the features is to add a dependency on the `spring-boot-starter-actuator` "Starter".

> **Definition of Actuator**
>
> An actuator is a manufacturing term that refers to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change.

To add the actuator to a Maven-based project, add the following "Starter" dependency:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

For Gradle, use the following declaration:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
}
```

13.2. Endpoints

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the `health` endpoint provides basic application health information.

You can enable or disable each individual endpoint and expose them (make them remotely accessible) over HTTP or JMX. An endpoint is considered to be available when it is both enabled and exposed. The built-in endpoints are auto-configured only when they are available. Most applications choose exposure over HTTP, where the ID of the endpoint and a prefix of `/actuator` is mapped to a URL. For example, by default, the `health` endpoint is mapped to `/actuator/health`.

TIP	To learn more about the Actuator's endpoints and their request and response formats, see the separate API documentation (HTML or PDF).

The following technology-agnostic endpoints are available:

ID	Description
`auditevents`	Exposes audit events information for the current application. Requires an `AuditEventRepository` bean.
`beans`	Displays a complete list of all the Spring beans in your application.
`caches`	Exposes available caches.
`conditions`	Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.
`configprops`	Displays a collated list of all `@ConfigurationProperties`. Subject to sanitization.
`env`	Exposes properties from Spring's `ConfigurableEnvironment`. Subject to sanitization.
`flyway`	Shows any Flyway database migrations that have been applied. Requires one or more `Flyway` beans.
`health`	Shows application health information.
`httpexchanges`	Displays HTTP exchange information (by default, the last 100 HTTP request-response exchanges). Requires an `HttpExchangeRepository` bean.
`info`	Displays arbitrary application info.
`integrationgraph`	Shows the Spring Integration graph. Requires a dependency on `spring-integration-core`.
`loggers`	Shows and modifies the configuration of loggers in the application.
`liquibase`	Shows any Liquibase database migrations that have been applied. Requires one or more `Liquibase` beans.
`metrics`	Shows "metrics" information for the current application.
`mappings`	Displays a collated list of all `@RequestMapping` paths.
`quartz`	Shows information about Quartz Scheduler jobs. Subject to sanitization.
`scheduledtasks`	Displays the scheduled tasks in your application.
`sessions`	Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Requires a servlet-based web application that uses Spring Session.
`shutdown`	Lets the application be gracefully shutdown. Only works when using jar packaging. Disabled by default.

ID	Description
`startup`	Shows the startup steps data collected by the `ApplicationStartup`. Requires the `SpringApplication` to be configured with a `BufferingApplicationStartup`.
`threaddump`	Performs a thread dump.

If your application is a web application (Spring MVC, Spring WebFlux, or Jersey), you can use the following additional endpoints:

ID	Description
`heapdump`	Returns a heap dump file. On a HotSpot JVM, an `HPROF`-format file is returned. On an OpenJ9 JVM, a `PHD`-format file is returned.
`logfile`	Returns the contents of the logfile (if the `logging.file.name` or the `logging.file.path` property has been set). Supports the use of the HTTP `Range` header to retrieve part of the log file's content.
`prometheus`	Exposes metrics in a format that can be scraped by a Prometheus server. Requires a dependency on `micrometer-registry-prometheus`.

13.2.1. Enabling Endpoints

By default, all endpoints except for `shutdown` are enabled. To configure the enablement of an endpoint, use its `management.endpoint.<id>.enabled` property. The following example enables the `shutdown` endpoint:

Properties

```
management.endpoint.shutdown.enabled=true
```

Yaml

```
management:
  endpoint:
    shutdown:
      enabled: true
```

If you prefer endpoint enablement to be opt-in rather than opt-out, set the `management.endpoints.enabled-by-default` property to `false` and use individual endpoint `enabled` properties to opt back in. The following example enables the `info` endpoint and disables all other endpoints:

Properties

```
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
```

Yaml

```
management:
  endpoints:
    enabled-by-default: false
  endpoint:
    info:
      enabled: true
```

NOTE	Disabled endpoints are removed entirely from the application context. If you want to change only the technologies over which an endpoint is exposed, use the `include` and `exclude` properties instead.

13.2.2. Exposing Endpoints

By default, only the health endpoint is exposed over HTTP and JMX. Since Endpoints may contain sensitive information, you should carefully consider when to expose them.

To change which endpoints are exposed, use the following technology-specific `include` and `exclude` properties:

Property	Default
`management.endpoints.jmx.exposure.exclude`	
`management.endpoints.jmx.exposure.include`	`health`
`management.endpoints.web.exposure.exclude`	
`management.endpoints.web.exposure.include`	`health`

The `include` property lists the IDs of the endpoints that are exposed. The `exclude` property lists the IDs of the endpoints that should not be exposed. The `exclude` property takes precedence over the `include` property. You can configure both the `include` and the `exclude` properties with a list of endpoint IDs.

For example, to only expose the `health` and `info` endpoints over JMX, use the following property:

Properties

```
management.endpoints.jmx.exposure.include=health,info
```

Yaml

```
management:
  endpoints:
    jmx:
      exposure:
        include: "health,info"
```

`*` can be used to select all endpoints. For example, to expose everything over HTTP except the `env`

and `beans` endpoints, use the following properties:

Properties

```
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=env,beans
```

Yaml

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
        exclude: "env,beans"
```

NOTE	`*` has a special meaning in YAML, so be sure to add quotation marks if you want to include (or exclude) all endpoints.
NOTE	If your application is exposed publicly, we strongly recommend that you also secure your endpoints.
TIP	If you want to implement your own strategy for when endpoints are exposed, you can register an `EndpointFilter` bean.

13.2.3. Security

For security purposes, only the `/health` endpoint is exposed over HTTP by default. You can use the `management.endpoints.web.exposure.include` property to configure the endpoints that are exposed.

NOTE	Before setting the `management.endpoints.web.exposure.include`, ensure that the exposed actuators do not contain sensitive information, are secured by placing them behind a firewall, or are secured by something like Spring Security.

If Spring Security is on the classpath and no other `SecurityFilterChain` bean is present, all actuators other than `/health` are secured by Spring Boot auto-configuration. If you define a custom `SecurityFilterChain` bean, Spring Boot auto-configuration backs off and lets you fully control the actuator access rules.

If you wish to configure custom security for HTTP endpoints (for example, to allow only users with a certain role to access them), Spring Boot provides some convenient `RequestMatcher` objects that you can use in combination with Spring Security.

A typical Spring Security configuration might look something like the following example:

Java

```
import
org.springframework.boot.actuate.autoconfigure.security.servlet.EndpointRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration(proxyBeanMethods = false)
public class MySecurityConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http.securityMatcher(EndpointRequest.toAnyEndpoint());
        http.authorizeHttpRequests((requests) ->
requests.anyRequest().hasRole("ENDPOINT_ADMIN"));
        http.httpBasic(withDefaults());
        return http.build();
    }

}
```

Kotlin

```
import org.springframework.boot.actuate.autoconfigure.security.servlet.EndpointRequest
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.config.Customizer.withDefaults
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import org.springframework.security.web.SecurityFilterChain

@Configuration(proxyBeanMethods = false)
class MySecurityConfiguration {

    @Bean
    fun securityFilterChain(http: HttpSecurity): SecurityFilterChain {
        http.securityMatcher(EndpointRequest.toAnyEndpoint()).authorizeHttpRequests {
requests ->
            requests.anyRequest().hasRole("ENDPOINT_ADMIN")
        }
        http.httpBasic(withDefaults())
        return http.build()
    }

}
```

The preceding example uses `EndpointRequest.toAnyEndpoint()` to match a request to any endpoint and then ensures that all have the `ENDPOINT_ADMIN` role. Several other matcher methods are also available on `EndpointRequest`. See the API documentation (HTML or PDF) for details.

If you deploy applications behind a firewall, you may prefer that all your actuator endpoints can be accessed without requiring authentication. You can do so by changing the `management.endpoints.web.exposure.include` property, as follows:

Properties

```
management.endpoints.web.exposure.include=*
```

Yaml

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

Additionally, if Spring Security is present, you would need to add custom security configuration that allows unauthenticated access to the endpoints, as the following example shows:

Java

```
import
org.springframework.boot.actuate.autoconfigure.security.servlet.EndpointRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration(proxyBeanMethods = false)
public class MySecurityConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http.securityMatcher(EndpointRequest.toAnyEndpoint());
        http.authorizeHttpRequests((requests) -> requests.anyRequest().permitAll());
        return http.build();
    }

}
```

Kotlin

```
import org.springframework.boot.actuate.autoconfigure.security.servlet.EndpointRequest
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import org.springframework.security.web.SecurityFilterChain

@Configuration(proxyBeanMethods = false)
class MySecurityConfiguration {

    @Bean
    fun securityFilterChain(http: HttpSecurity): SecurityFilterChain {
        http.securityMatcher(EndpointRequest.toAnyEndpoint()).authorizeHttpRequests {
requests ->
            requests.anyRequest().permitAll()
        }
        return http.build()
    }

}
```

NOTE	In both of the preceding examples, the configuration applies only to the actuator endpoints. Since Spring Boot's security configuration backs off completely in the presence of any `SecurityFilterChain` bean, you need to configure an additional `SecurityFilterChain` bean with rules that apply to the rest of the application.

Cross Site Request Forgery Protection

Since Spring Boot relies on Spring Security's defaults, CSRF protection is turned on by default. This means that the actuator endpoints that require a `POST` (shutdown and loggers endpoints), a `PUT`, or a `DELETE` get a 403 (forbidden) error when the default security configuration is in use.

NOTE	We recommend disabling CSRF protection completely only if you are creating a service that is used by non-browser clients.

You can find additional information about CSRF protection in the Spring Security Reference Guide.

13.2.4. Configuring Endpoints

Endpoints automatically cache responses to read operations that do not take any parameters. To configure the amount of time for which an endpoint caches a response, use its `cache.time-to-live` property. The following example sets the time-to-live of the `beans` endpoint's cache to 10 seconds:

Properties

```
management.endpoint.beans.cache.time-to-live=10s
```

Yaml

```
management:
  endpoint:
    beans:
      cache:
        time-to-live: "10s"
```

NOTE The `management.endpoint.<name>` prefix uniquely identifies the endpoint that is being configured.

13.2.5. Sanitize Sensitive Values

Information returned by the `/env`, `/configprops` and `/quartz` endpoints can be somewhat sensitive. All values are sanitized by default (that is replaced by `******`). Viewing original values in the unsanitized form can be configured per endpoint using the `showValues` property for that endpoint. This property can be configured to have the following values:

- `ALWAYS` - all values are shown in their unsanitized form to all users
- `NEVER` - all values are always sanitized (that is replaced by `******`)
- `WHEN_AUTHORIZED` - all values are shown in their unsanitized form to authorized users

For HTTP endpoints, a user is considered to be authorized if they have authenticated and have the roles configured by the endpoint's roles property. By default, any authenticated user is authorized. For JMX endpoints, all users are always authorized.

Properties

```
management.endpoint.env.show-values=WHEN_AUTHORIZED
management.endpoint.env.roles=admin
```

Yaml

```
management:
  endpoint:
    env:
      show-values: WHEN_AUTHORIZED
      roles: "admin"
```

The configuration above enables the ability for all users with the `admin` role to view all values in their original form from the `/env` endpoint.

NOTE When `show-values` is set to `ALWAYS` or `WHEN_AUTHORIZED` any sanitization applied by a `SanitizingFunction` will still be applied.

13.2.6. Hypermedia for Actuator Web Endpoints

A “discovery page” is added with links to all the endpoints. The “discovery page” is available on `/actuator` by default.

To disable the “discovery page”, add the following property to your application properties:

Properties

```
management.endpoints.web.discovery.enabled=false
```

Yaml

```
management:
  endpoints:
    web:
      discovery:
        enabled: false
```

When a custom management context path is configured, the “discovery page” automatically moves from `/actuator` to the root of the management context. For example, if the management context path is `/management`, the discovery page is available from `/management`. When the management context path is set to `/`, the discovery page is disabled to prevent the possibility of a clash with other mappings.

13.2.7. CORS Support

Cross-origin resource sharing (CORS) is a W3C specification that lets you specify in a flexible way what kind of cross-domain requests are authorized. If you use Spring MVC or Spring WebFlux, you can configure Actuator’s web endpoints to support such scenarios.

CORS support is disabled by default and is only enabled once you have set the `management.endpoints.web.cors.allowed-origins` property. The following configuration permits `GET` and `POST` calls from the `example.com` domain:

Properties

```
management.endpoints.web.cors.allowed-origins=https://example.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

Yaml

```
management:
  endpoints:
    web:
      cors:
        allowed-origins: "https://example.com"
        allowed-methods: "GET,POST"
```

TIP | See `CorsEndpointProperties` for a complete list of options.

13.2.8. Implementing Custom Endpoints

If you add a `@Bean` annotated with `@Endpoint`, any methods annotated with `@ReadOperation`, `@WriteOperation`, or `@DeleteOperation` are automatically exposed over JMX and, in a web application, over HTTP as well. Endpoints can be exposed over HTTP by using Jersey, Spring MVC, or Spring WebFlux. If both Jersey and Spring MVC are available, Spring MVC is used.

The following example exposes a read operation that returns a custom object:

Java

```
@ReadOperation
public CustomData getData() {
    return new CustomData("test", 5);
}
```

Kotlin

```
@ReadOperation
fun getData(): CustomData {
    return CustomData("test", 5)
}
```

You can also write technology-specific endpoints by using `@JmxEndpoint` or `@WebEndpoint`. These endpoints are restricted to their respective technologies. For example, `@WebEndpoint` is exposed only over HTTP and not over JMX.

You can write technology-specific extensions by using `@EndpointWebExtension` and `@EndpointJmxExtension`. These annotations let you provide technology-specific operations to augment an existing endpoint.

Finally, if you need access to web-framework-specific functionality, you can implement servlet or Spring `@Controller` and `@RestController` endpoints at the cost of them not being available over JMX or when using a different web framework.

Receiving Input

Operations on an endpoint receive input through their parameters. When exposed over the web, the values for these parameters are taken from the URL's query parameters and from the JSON request body. When exposed over JMX, the parameters are mapped to the parameters of the MBean's operations. Parameters are required by default. They can be made optional by annotating them with either `@javax.annotation.Nullable` or `@org.springframework.lang.Nullable`.

You can map each root property in the JSON request body to a parameter of the endpoint. Consider the following JSON request body:

```
{
    "name": "test",
    "counter": 42
}
```

You can use this to invoke a write operation that takes `String name` and `int counter` parameters, as the following example shows:

Java

```
@WriteOperation
public void updateData(String name, int counter) {
    // injects "test" and 42
}
```

Kotlin

```
@WriteOperation
fun updateData(name: String?, counter: Int) {
    // injects "test" and 42
}
```

TIP	Because endpoints are technology agnostic, only simple types can be specified in the method signature. In particular, declaring a single parameter with a `CustomData` type that defines a `name` and `counter` properties is not supported.

NOTE	To let the input be mapped to the operation method's parameters, Java code that implements an endpoint should be compiled with `-parameters`, and Kotlin code that implements an endpoint should be compiled with `-java-parameters`. This will happen automatically if you use Spring Boot's Gradle plugin or if you use Maven and `spring-boot-starter-parent`.

Input Type Conversion

The parameters passed to endpoint operation methods are, if necessary, automatically converted to the required type. Before calling an operation method, the input received over JMX or HTTP is converted to the required types by using an instance of `ApplicationConversionService` as well as any `Converter` or `GenericConverter` beans qualified with `@EndpointConverter`.

Custom Web Endpoints

Operations on an `@Endpoint`, `@WebEndpoint`, or `@EndpointWebExtension` are automatically exposed over HTTP using Jersey, Spring MVC, or Spring WebFlux. If both Jersey and Spring MVC are available, Spring MVC is used.

Web Endpoint Request Predicates

A request predicate is automatically generated for each operation on a web-exposed endpoint.

Path

The path of the predicate is determined by the ID of the endpoint and the base path of the web-exposed endpoints. The default base path is `/actuator`. For example, an endpoint with an ID of `sessions` uses `/actuator/sessions` as its path in the predicate.

You can further customize the path by annotating one or more parameters of the operation method with `@Selector`. Such a parameter is added to the path predicate as a path variable. The variable's value is passed into the operation method when the endpoint operation is invoked. If you want to capture all remaining path elements, you can add `@Selector(Match=ALL_REMAINING)` to the last parameter and make it a type that is conversion-compatible with a `String[]`.

HTTP method

The HTTP method of the predicate is determined by the operation type, as shown in the following table:

Operation	HTTP method
`@ReadOperation`	`GET`
`@WriteOperation`	`POST`
`@DeleteOperation`	`DELETE`

Consumes

For a `@WriteOperation` (HTTP `POST`) that uses the request body, the `consumes` clause of the predicate is `application/vnd.spring-boot.actuator.v2+json, application/json`. For all other operations, the `consumes` clause is empty.

Produces

The `produces` clause of the predicate can be determined by the `produces` attribute of the `@DeleteOperation`, `@ReadOperation`, and `@WriteOperation` annotations. The attribute is optional. If it is not used, the `produces` clause is determined automatically.

If the operation method returns `void` or `Void`, the `produces` clause is empty. If the operation method returns a `org.springframework.core.io.Resource`, the `produces` clause is `application/octet-stream`. For all other operations, the `produces` clause is `application/vnd.spring-boot.actuator.v2+json, application/json`.

Web Endpoint Response Status

The default response status for an endpoint operation depends on the operation type (read, write, or delete) and what, if anything, the operation returns.

If a `@ReadOperation` returns a value, the response status will be 200 (OK). If it does not return a value, the response status will be 404 (Not Found).

If a `@WriteOperation` or `@DeleteOperation` returns a value, the response status will be 200 (OK). If it does not return a value, the response status will be 204 (No Content).

If an operation is invoked without a required parameter or with a parameter that cannot be converted to the required type, the operation method is not called, and the response status will be 400 (Bad Request).

Web Endpoint Range Requests

You can use an HTTP range request to request part of an HTTP resource. When using Spring MVC or Spring Web Flux, operations that return a `org.springframework.core.io.Resource` automatically support range requests.

NOTE	Range requests are not supported when using Jersey.

Web Endpoint Security

An operation on a web endpoint or a web-specific endpoint extension can receive the current `java.security.Principal` or `org.springframework.boot.actuate.endpoint.SecurityContext` as a method parameter. The former is typically used in conjunction with `@Nullable` to provide different behavior for authenticated and unauthenticated users. The latter is typically used to perform authorization checks by using its `isUserInRole(String)` method.

Servlet Endpoints

A servlet can be exposed as an endpoint by implementing a class annotated with `@ServletEndpoint` that also implements `Supplier<EndpointServlet>`. Servlet endpoints provide deeper integration with the servlet container but at the expense of portability. They are intended to be used to expose an existing servlet as an endpoint. For new endpoints, the `@Endpoint` and `@WebEndpoint` annotations should be preferred whenever possible.

Controller Endpoints

You can use `@ControllerEndpoint` and `@RestControllerEndpoint` to implement an endpoint that is exposed only by Spring MVC or Spring WebFlux. Methods are mapped by using the standard annotations for Spring MVC and Spring WebFlux, such as `@RequestMapping` and `@GetMapping`, with the endpoint's ID being used as a prefix for the path. Controller endpoints provide deeper integration with Spring's web frameworks but at the expense of portability. The `@Endpoint` and `@WebEndpoint` annotations should be preferred whenever possible.

13.2.9. Health Information

You can use health information to check the status of your running application. It is often used by monitoring software to alert someone when a production system goes down. The information exposed by the `health` endpoint depends on the `management.endpoint.health.show-details` and `management.endpoint.health.show-components` properties, which can be configured with one of the following values:

Name	Description
`never`	Details are never shown.
`when-authorized`	Details are shown only to authorized users. Authorized roles can be configured by using `management.endpoint.health.roles`.
`always`	Details are shown to all users.

The default value is `never`. A user is considered to be authorized when they are in one or more of the endpoint's roles. If the endpoint has no configured roles (the default), all authenticated users are considered to be authorized. You can configure the roles by using the `management.endpoint.health.roles` property.

NOTE If you have secured your application and wish to use `always`, your security configuration must permit access to the health endpoint for both authenticated and unauthenticated users.

Health information is collected from the content of a `HealthContributorRegistry` (by default, all `HealthContributor` instances defined in your `ApplicationContext`). Spring Boot includes a number of auto-configured `HealthContributors`, and you can also write your own.

A `HealthContributor` can be either a `HealthIndicator` or a `CompositeHealthContributor`. A `HealthIndicator` provides actual health information, including a `Status`. A `CompositeHealthContributor` provides a composite of other `HealthContributors`. Taken together, contributors form a tree structure to represent the overall system health.

By default, the final system health is derived by a `StatusAggregator`, which sorts the statuses from each `HealthIndicator` based on an ordered list of statuses. The first status in the sorted list is used as the overall health status. If no `HealthIndicator` returns a status that is known to the `StatusAggregator`, an `UNKNOWN` status is used.

TIP You can use the `HealthContributorRegistry` to register and unregister health indicators at runtime.

Auto-configured HealthIndicators

When appropriate, Spring Boot auto-configures the `HealthIndicators` listed in the following table. You can also enable or disable selected indicators by configuring `management.health.key.enabled`, with the `key` listed in the following table:

Key	Name	Description
`cassandra`	`CassandraDriverHealthIndicator`	Checks that a Cassandra database is up.
`couchbase`	`CouchbaseHealthIndicator`	Checks that a Couchbase cluster is up.
`db`	`DataSourceHealthIndicator`	Checks that a connection to `DataSource` can be obtained.
`diskspace`	`DiskSpaceHealthIndicator`	Checks for low disk space.

Key	Name	Description
elasticsearch	ElasticsearchRestClientHealthIndicator	Checks that an Elasticsearch cluster is up.
hazelcast	HazelcastHealthIndicator	Checks that a Hazelcast server is up.
influxdb	InfluxDbHealthIndicator	Checks that an InfluxDB server is up.
jms	JmsHealthIndicator	Checks that a JMS broker is up.
ldap	LdapHealthIndicator	Checks that an LDAP server is up.
mail	MailHealthIndicator	Checks that a mail server is up.
mongo	MongoHealthIndicator	Checks that a Mongo database is up.
neo4j	Neo4jHealthIndicator	Checks that a Neo4j database is up.
ping	PingHealthIndicator	Always responds with UP.
rabbit	RabbitHealthIndicator	Checks that a Rabbit server is up.
redis	RedisHealthIndicator	Checks that a Redis server is up.

TIP You can disable them all by setting the `management.health.defaults.enabled` property.

Additional `HealthIndicators` are available but are not enabled by default:

Key	Name	Description
livenessstate	LivenessStateHealthIndicator	Exposes the “Liveness” application availability state.
readinessstate	ReadinessStateHealthIndicator	Exposes the “Readiness” application availability state.

Writing Custom HealthIndicators

To provide custom health information, you can register Spring beans that implement the `HealthIndicator` interface. You need to provide an implementation of the `health()` method and return a `Health` response. The `Health` response should include a status and can optionally include additional details to be displayed. The following code shows a sample `HealthIndicator` implementation:

Java

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check();
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }

    private int check() {
        // perform some specific health check
        return ...
    }

}
```

Kotlin

```
import org.springframework.boot.actuate.health.Health
import org.springframework.boot.actuate.health.HealthIndicator
import org.springframework.stereotype.Component

@Component
class MyHealthIndicator : HealthIndicator {

    override fun health(): Health {
        val errorCode = check()
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build()
        }
        return Health.up().build()
    }

    private fun check(): Int {
        // perform some specific health check
        return  ...
    }

}
```

NOTE	The identifier for a given `HealthIndicator` is the name of the bean without the `HealthIndicator` suffix, if it exists. In the preceding example, the health information is available in an entry named `my`.

TIP	Health indicators are usually called over HTTP and need to respond before any connection timeouts. Spring Boot will log a warning message for any health indicator that takes longer than 10 seconds to respond. If you want to configure this threshold, you can use the `management.endpoint.health.logging.slow-indicator-threshold` property.

In addition to Spring Boot's predefined `Status` types, `Health` can return a custom `Status` that represents a new system state. In such cases, you also need to provide a custom implementation of the `StatusAggregator` interface, or you must configure the default implementation by using the `management.endpoint.health.status.order` configuration property.

For example, assume a new `Status` with a code of `FATAL` is being used in one of your `HealthIndicator` implementations. To configure the severity order, add the following property to your application properties:

Properties

```
management.endpoint.health.status.order=fatal,down,out-of-service,unknown,up
```

Yaml

```
management:
  endpoint:
    health:
      status:
        order: "fatal,down,out-of-service,unknown,up"
```

The HTTP status code in the response reflects the overall health status. By default, `OUT_OF_SERVICE` and `DOWN` map to 503. Any unmapped health statuses, including `UP`, map to 200. You might also want to register custom status mappings if you access the health endpoint over HTTP. Configuring a custom mapping disables the defaults mappings for `DOWN` and `OUT_OF_SERVICE`. If you want to retain the default mappings, you must explicitly configure them, alongside any custom mappings. For example, the following property maps `FATAL` to 503 (service unavailable) and retains the default mappings for `DOWN` and `OUT_OF_SERVICE`:

Properties

```
management.endpoint.health.status.http-mapping.down=503
management.endpoint.health.status.http-mapping.fatal=503
management.endpoint.health.status.http-mapping.out-of-service=503
```

Yaml

```
management:
  endpoint:
    health:
      status:
        http-mapping:
          down: 503
          fatal: 503
          out-of-service: 503
```

TIP If you need more control, you can define your own `HttpCodeStatusMapper` bean.

The following table shows the default status mappings for the built-in statuses:

Status	Mapping
`DOWN`	`SERVICE_UNAVAILABLE` (`503`)
`OUT_OF_SERVICE`	`SERVICE_UNAVAILABLE` (`503`)
`UP`	No mapping by default, so HTTP status is `200`
`UNKNOWN`	No mapping by default, so HTTP status is `200`

Reactive Health Indicators

For reactive applications, such as those that use Spring WebFlux, `ReactiveHealthContributor` provides a non-blocking contract for getting application health. Similar to a traditional `HealthContributor`, health information is collected from the content of a `ReactiveHealthContributorRegistry` (by default, all `HealthContributor` and `ReactiveHealthContributor` instances defined in your `ApplicationContext`). Regular `HealthContributors` that do not check against a reactive API are executed on the elastic scheduler.

TIP In a reactive application, you should use the `ReactiveHealthContributorRegistry` to register and unregister health indicators at runtime. If you need to register a regular `HealthContributor`, you should wrap it with `ReactiveHealthContributor#adapt`.

To provide custom health information from a reactive API, you can register Spring beans that implement the `ReactiveHealthIndicator` interface. The following code shows a sample `ReactiveHealthIndicator` implementation:

Java

```
import reactor.core.publisher.Mono;

import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.ReactiveHealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyReactiveHealthIndicator implements ReactiveHealthIndicator {

    @Override
    public Mono<Health> health() {
        return doHealthCheck().onErrorResume((exception) ->
            Mono.just(new Health.Builder().down(exception).build()));
    }

    private Mono<Health> doHealthCheck() {
        // perform some specific health check
        return ...
    }

}
```

Kotlin

```
import org.springframework.boot.actuate.health.Health
import org.springframework.boot.actuate.health.ReactiveHealthIndicator
import org.springframework.stereotype.Component
import reactor.core.publisher.Mono

@Component
class MyReactiveHealthIndicator : ReactiveHealthIndicator {

    override fun health(): Mono<Health> {
        return doHealthCheck()!!.onErrorResume { exception: Throwable? ->
            Mono.just(Health.Builder().down(exception).build())
        }
    }

    private fun doHealthCheck(): Mono<Health>? {
        // perform some specific health check
        return  ...
    }

}
```

TIP To handle the error automatically, consider extending from `AbstractReactiveHealthIndicator`.

Auto-configured ReactiveHealthIndicators

When appropriate, Spring Boot auto-configures the following `ReactiveHealthIndicators`:

Key	Name	Description
`cassandra`	`CassandraDriverReactiveHealthIndicator`	Checks that a Cassandra database is up.
`couchbase`	`CouchbaseReactiveHealthIndicator`	Checks that a Couchbase cluster is up.
`elasticsearch`	`ElasticsearchReactiveHealthIndicator`	Checks that an Elasticsearch cluster is up.
`mongo`	`MongoReactiveHealthIndicator`	Checks that a Mongo database is up.
`neo4j`	`Neo4jReactiveHealthIndicator`	Checks that a Neo4j database is up.
`redis`	`RedisReactiveHealthIndicator`	Checks that a Redis server is up.

TIP If necessary, reactive indicators replace the regular ones. Also, any `HealthIndicator` that is not handled explicitly is wrapped automatically.

Health Groups

It is sometimes useful to organize health indicators into groups that you can use for different purposes.

To create a health indicator group, you can use the `management.endpoint.health.group.<name>` property and specify a list of health indicator IDs to `include` or `exclude`. For example, to create a group that includes only database indicators you can define the following:

Properties

```
management.endpoint.health.group.custom.include=db
```

Yaml

```
management:
  endpoint:
    health:
      group:
        custom:
          include: "db"
```

You can then check the result by hitting `localhost:8080/actuator/health/custom`.

Similarly, to create a group that excludes the database indicators from the group and includes all the other indicators, you can define the following:

Properties

```
management.endpoint.health.group.custom.exclude=db
```

Yaml

```
management:
  endpoint:
    health:
      group:
        custom:
          exclude: "db"
```

By default, startup will fail if a health group includes or excludes a health indicator that does not exist. To disable this behavior set `management.endpoint.health.validate-group-membership` to `false`.

By default, groups inherit the same `StatusAggregator` and `HttpCodeStatusMapper` settings as the system health. However, you can also define these on a per-group basis. You can also override the `show-details` and `roles` properties if required:

Properties

```
management.endpoint.health.group.custom.show-details=when-authorized
management.endpoint.health.group.custom.roles=admin
management.endpoint.health.group.custom.status.order=fatal,up
management.endpoint.health.group.custom.status.http-mapping.fatal=500
management.endpoint.health.group.custom.status.http-mapping.out-of-service=500
```

Yaml

```
management:
  endpoint:
    health:
      group:
        custom:
          show-details: "when-authorized"
          roles: "admin"
          status:
            order: "fatal,up"
            http-mapping:
              fatal: 500
              out-of-service: 500
```

TIP You can use `@Qualifier("groupname")` if you need to register custom `StatusAggregator` or `HttpCodeStatusMapper` beans for use with the group.

A health group can also include/exclude a `CompositeHealthContributor`. You can also include/exclude only a certain component of a `CompositeHealthContributor`. This can be done using the fully

qualified name of the component as follows:

```
management.endpoint.health.group.custom.include="test/primary"
management.endpoint.health.group.custom.exclude="test/primary/b"
```

In the example above, the `custom` group will include the `HealthContributor` with the name `primary` which is a component of the composite `test`. Here, `primary` itself is a composite and the `HealthContributor` with the name `b` will be excluded from the `custom` group.

Health groups can be made available at an additional path on either the main or management port. This is useful in cloud environments such as Kubernetes, where it is quite common to use a separate management port for the actuator endpoints for security purposes. Having a separate port could lead to unreliable health checks because the main application might not work properly even if the health check is successful. The health group can be configured with an additional path as follows:

```
management.endpoint.health.group.live.additional-path="server:/healthz"
```

This would make the `live` health group available on the main server port at `/healthz`. The prefix is mandatory and must be either `server:` (represents the main server port) or `management:` (represents the management port, if configured.) The path must be a single path segment.

DataSource Health

The `DataSource` health indicator shows the health of both standard data sources and routing data source beans. The health of a routing data source includes the health of each of its target data sources. In the health endpoint's response, each of a routing data source's targets is named by using its routing key. If you prefer not to include routing data sources in the indicator's output, set `management.health.db.ignore-routing-data-sources` to `true`.

13.2.10. Kubernetes Probes

Applications deployed on Kubernetes can provide information about their internal state with Container Probes. Depending on your Kubernetes configuration, the kubelet calls those probes and reacts to the result.

By default, Spring Boot manages your Application Availability State. If deployed in a Kubernetes environment, actuator gathers the "Liveness" and "Readiness" information from the `ApplicationAvailability` interface and uses that information in dedicated health indicators: `LivenessStateHealthIndicator` and `ReadinessStateHealthIndicator`. These indicators are shown on the global health endpoint (`"/actuator/health"`). They are also exposed as separate HTTP Probes by using health groups: `"/actuator/health/liveness"` and `"/actuator/health/readiness"`.

You can then configure your Kubernetes infrastructure with the following endpoint information:

```
livenessProbe:
  httpGet:
    path: "/actuator/health/liveness"
    port: <actuator-port>
  failureThreshold: ...
  periodSeconds: ...

readinessProbe:
  httpGet:
    path: "/actuator/health/readiness"
    port: <actuator-port>
  failureThreshold: ...
  periodSeconds: ...
```

NOTE `<actuator-port>` should be set to the port that the actuator endpoints are available on. It could be the main web server port or a separate management port if the `"management.server.port"` property has been set.

These health groups are automatically enabled only if the application runs in a Kubernetes environment. You can enable them in any environment by using the `management.endpoint.health.probes.enabled` configuration property.

NOTE If an application takes longer to start than the configured liveness period, Kubernetes mentions the `"startupProbe"` as a possible solution. Generally speaking, the `"startupProbe"` is not necessarily needed here, as the `"readinessProbe"` fails until all startup tasks are done. This means your application will not receive traffic until it is ready. However, if your application takes a long time to start, consider using a `"startupProbe"` to make sure that Kubernetes won't kill your application while it is in the process of starting. See the section that describes how probes behave during the application lifecycle.

If your Actuator endpoints are deployed on a separate management context, the endpoints do not use the same web infrastructure (port, connection pools, framework components) as the main application. In this case, a probe check could be successful even if the main application does not work properly (for example, it cannot accept new connections). For this reason, is it a good idea to make the `liveness` and `readiness` health groups available on the main server port. This can be done by setting the following property:

```
management.endpoint.health.probes.add-additional-paths=true
```

This would make the `liveness` group available at `/livez` and the `readiness` group available at `/readyz` on the main server port. Paths can be customized using the `additional-path` property on each group, see health groups for details.

Checking External State With Kubernetes Probes

Actuator configures the “liveness” and “readiness” probes as Health Groups. This means that all the health groups features are available for them. You can, for example, configure additional Health Indicators:

Properties

```
management.endpoint.health.group.readiness.include=readinessState,customCheck
```

Yaml

```
management:
  endpoint:
    health:
      group:
        readiness:
          include: "readinessState,customCheck"
```

By default, Spring Boot does not add other health indicators to these groups.

The “liveness” probe should not depend on health checks for external systems. If the liveness state of an application is broken, Kubernetes tries to solve that problem by restarting the application instance. This means that if an external system (such as a database, a Web API, or an external cache) fails, Kubernetes might restart all application instances and create cascading failures.

As for the “readiness” probe, the choice of checking external systems must be made carefully by the application developers. For this reason, Spring Boot does not include any additional health checks in the readiness probe. If the readiness state of an application instance is unready, Kubernetes does not route traffic to that instance. Some external systems might not be shared by application instances, in which case they could be included in a readiness probe. Other external systems might not be essential to the application (the application could have circuit breakers and fallbacks), in which case they definitely should not be included. Unfortunately, an external system that is shared by all application instances is common, and you have to make a judgement call: Include it in the readiness probe and expect that the application is taken out of service when the external service is down or leave it out and deal with failures higher up the stack, perhaps by using a circuit breaker in the caller.

NOTE

If all instances of an application are unready, a Kubernetes Service with `type=ClusterIP` or `NodePort` does not accept any incoming connections. There is no HTTP error response (503 and so on), since there is no connection. A service with `type=LoadBalancer` might or might not accept connections, depending on the provider. A service that has an explicit ingress also responds in a way that depends on the implementation — the ingress service itself has to decide how to handle the “connection refused” from downstream. HTTP 503 is quite likely in the case of both load balancer and ingress.

Also, if an application uses Kubernetes autoscaling, it may react differently to applications being

taken out of the load-balancer, depending on its autoscaler configuration.

Application Lifecycle and Probe States

An important aspect of the Kubernetes Probes support is its consistency with the application lifecycle. There is a significant difference between the `AvailabilityState` (which is the in-memory, internal state of the application) and the actual probe (which exposes that state). Depending on the phase of application lifecycle, the probe might not be available.

Spring Boot publishes application events during startup and shutdown, and probes can listen to such events and expose the `AvailabilityState` information.

The following tables show the `AvailabilityState` and the state of HTTP connectors at different stages.

When a Spring Boot application starts:

Startup phase	LivenessState	ReadinessState	HTTP server	Notes
Starting	`BROKEN`	`REFUSING_TRAFFIC`	Not started	Kubernetes checks the "liveness" Probe and restarts the application if it takes too long.
Started	`CORRECT`	`REFUSING_TRAFFIC`	Refuses requests	The application context is refreshed. The application performs startup tasks and does not receive traffic yet.
Ready	`CORRECT`	`ACCEPTING_TRAFFIC`	Accepts requests	Startup tasks are finished. The application is receiving traffic.

When a Spring Boot application shuts down:

Shutdown phase	Liveness State	Readiness State	HTTP server	Notes
Running	`CORRECT`	`ACCEPTING_TRAFFIC`	Accepts requests	Shutdown has been requested.
Graceful shutdown	`CORRECT`	`REFUSING_TRAFFIC`	New requests are rejected	If enabled, graceful shutdown processes in-flight requests.
Shutdown complete	N/A	N/A	Server is shut down	The application context is closed and the application is shut down.

TIP See Kubernetes container lifecycle section for more information about Kubernetes deployment.

13.2.11. Application Information

Application information exposes various information collected from all `InfoContributor` beans defined in your `ApplicationContext`. Spring Boot includes a number of auto-configured

`InfoContributor` beans, and you can write your own.

Auto-configured InfoContributors

When appropriate, Spring auto-configures the following `InfoContributor` beans:

ID	Name	Description	Prerequisites
`build`	`BuildInfoContributor`	Exposes build information.	A `META-INF/build-info.properties` resource.
`env`	`EnvironmentInfoContributor`	Exposes any property from the `Environment` whose name starts with `info.`.	None.
`git`	`GitInfoContributor`	Exposes git information.	A `git.properties` resource.
`java`	`JavaInfoContributor`	Exposes Java runtime information.	None.
`os`	`OsInfoContributor`	Exposes Operating System information.	None.

Whether an individual contributor is enabled is controlled by its `management.info.<id>.enabled` property. Different contributors have different defaults for this property, depending on their prerequisites and the nature of the information that they expose.

With no prerequisites to indicate that they should be enabled, the `env`, `java`, and `os` contributors are disabled by default. Each can be enabled by setting its `management.info.<id>.enabled` property to `true`.

The `build` and `git` info contributors are enabled by default. Each can be disabled by setting its `management.info.<id>.enabled` property to `false`. Alternatively, to disable every contributor that is usually enabled by default, set the `management.info.defaults.enabled` property to `false`.

Custom Application Information

When the `env` contributor is enabled, you can customize the data exposed by the `info` endpoint by setting `info.*` Spring properties. All `Environment` properties under the `info` key are automatically exposed. For example, you could add the following settings to your `application.properties` file:

Properties

```
info.app.encoding=UTF-8
info.app.java.source=17
info.app.java.target=17
```

Yaml

```
info:
  app:
    encoding: "UTF-8"
    java:
      source: "17"
      target: "17"
```

TIP

Rather than hardcoding those values, you could also expand info properties at build time.

Assuming you use Maven, you could rewrite the preceding example as follows:

Properties

```
info.app.encoding=@project.build.sourceEncoding@
info.app.java.source=@java.version@
info.app.java.target=@java.version@
```

Yaml

```
info:
  app:
    encoding: "@project.build.sourceEncoding@"
    java:
      source: "@java.version@"
      target: "@java.version@"
```

Git Commit Information

Another useful feature of the `info` endpoint is its ability to publish information about the state of your `git` source code repository when the project was built. If a `GitProperties` bean is available, you can use the `info` endpoint to expose these properties.

TIP

A `GitProperties` bean is auto-configured if a `git.properties` file is available at the root of the classpath. See "how to generate git information" for more detail.

By default, the endpoint exposes `git.branch`, `git.commit.id`, and `git.commit.time` properties, if present. If you do not want any of these properties in the endpoint response, they need to be excluded from the `git.properties` file. If you want to display the full git information (that is, the full content of `git.properties`), use the `management.info.git.mode` property, as follows:

Properties

```
management.info.git.mode=full
```

Yaml

```
management:
  info:
    git:
      mode: "full"
```

To disable the git commit information from the `info` endpoint completely, set the `management.info.git.enabled` property to `false`, as follows:

Properties

```
management.info.git.enabled=false
```

Yaml

```
management:
  info:
    git:
      enabled: false
```

Build Information

If a `BuildProperties` bean is available, the `info` endpoint can also publish information about your build. This happens if a `META-INF/build-info.properties` file is available in the classpath.

TIP	The Maven and Gradle plugins can both generate that file. See "how to generate build information" for more details.

Java Information

The `info` endpoint publishes information about your Java runtime environment, see `JavaInfo` for more details.

OS Information

The `info` endpoint publishes information about your Operating System, see `OsInfo` for more details.

Writing Custom InfoContributors

To provide custom application information, you can register Spring beans that implement the `InfoContributor` interface.

The following example contributes an `example` entry with a single value:

Java

```
import java.util.Collections;

import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class MyInfoContributor implements InfoContributor {

    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("example", Collections.singletonMap("key", "value"));
    }

}
```

Kotlin

```
import org.springframework.boot.actuate.info.Info
import org.springframework.boot.actuate.info.InfoContributor
import org.springframework.stereotype.Component
import java.util.Collections

@Component
class MyInfoContributor : InfoContributor {

    override fun contribute(builder: Info.Builder) {
        builder.withDetail("example", Collections.singletonMap("key", "value"))
    }

}
```

If you reach the `info` endpoint, you should see a response that contains the following additional entry:

```
{
    "example": {
        "key" : "value"
    }
}
```

13.3. Monitoring and Management Over HTTP

If you are developing a web application, Spring Boot Actuator auto-configures all enabled endpoints to be exposed over HTTP. The default convention is to use the `id` of the endpoint with a prefix of `/actuator` as the URL path. For example, `health` is exposed as `/actuator/health`.

TIP	Actuator is supported natively with Spring MVC, Spring WebFlux, and Jersey. If both Jersey and Spring MVC are available, Spring MVC is used.

NOTE	Jackson is a required dependency in order to get the correct JSON responses as documented in the API documentation (HTML or PDF).

13.3.1. Customizing the Management Endpoint Paths

Sometimes, it is useful to customize the prefix for the management endpoints. For example, your application might already use `/actuator` for another purpose. You can use the `management.endpoints.web.base-path` property to change the prefix for your management endpoint, as the following example shows:

Properties

```
management.endpoints.web.base-path=/manage
```

Yaml

```
management:
  endpoints:
    web:
      base-path: "/manage"
```

The preceding `application.properties` example changes the endpoint from `/actuator/{id}` to `/manage/{id}` (for example, `/manage/info`).

NOTE	Unless the management port has been configured to expose endpoints by using a different HTTP port, `management.endpoints.web.base-path` is relative to `server.servlet.context-path` (for servlet web applications) or `spring.webflux.base-path` (for reactive web applications). If `management.server.port` is configured, `management.endpoints.web.base-path` is relative to `management.server.base-path`.

If you want to map endpoints to a different path, you can use the `management.endpoints.web.path-mapping` property.

The following example remaps `/actuator/health` to `/healthcheck`:

Properties

```
management.endpoints.web.base-path=/
management.endpoints.web.path-mapping.health=healthcheck
```

Yaml

```
management:
  endpoints:
    web:
      base-path: "/"
      path-mapping:
        health: "healthcheck"
```

13.3.2. Customizing the Management Server Port

Exposing management endpoints by using the default HTTP port is a sensible choice for cloud-based deployments. If, however, your application runs inside your own data center, you may prefer to expose endpoints by using a different HTTP port.

You can set the `management.server.port` property to change the HTTP port, as the following example shows:

Properties

```
management.server.port=8081
```

Yaml

```
management:
  server:
    port: 8081
```

NOTE On Cloud Foundry, by default, applications receive requests only on port 8080 for both HTTP and TCP routing. If you want to use a custom management port on Cloud Foundry, you need to explicitly set up the application's routes to forward traffic to the custom port.

13.3.3. Configuring Management-specific SSL

When configured to use a custom port, you can also configure the management server with its own SSL by using the various `management.server.ssl.*` properties. For example, doing so lets a management server be available over HTTP while the main application uses HTTPS, as the following property settings show:

Properties

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:store.jks
server.ssl.key-password=secret
management.server.port=8080
management.server.ssl.enabled=false
```

Yaml

```
server:
  port: 8443
  ssl:
    enabled: true
    key-store: "classpath:store.jks"
    key-password: "secret"
management:
  server:
    port: 8080
    ssl:
      enabled: false
```

Alternatively, both the main server and the management server can use SSL but with different key stores, as follows:

Properties

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:main.jks
server.ssl.key-password=secret
management.server.port=8080
management.server.ssl.enabled=true
management.server.ssl.key-store=classpath:management.jks
management.server.ssl.key-password=secret
```

Yaml

```
server:
  port: 8443
  ssl:
    enabled: true
    key-store: "classpath:main.jks"
    key-password: "secret"
management:
  server:
    port: 8080
    ssl:
      enabled: true
      key-store: "classpath:management.jks"
      key-password: "secret"
```

13.3.4. Customizing the Management Server Address

You can customize the address on which the management endpoints are available by setting the `management.server.address` property. Doing so can be useful if you want to listen only on an internal or ops-facing network or to listen only for connections from `localhost`.

NOTE	You can listen on a different address only when the port differs from the main server port.

The following example `application.properties` does not allow remote management connections:

Properties

```
management.server.port=8081
management.server.address=127.0.0.1
```

Yaml

```
management:
  server:
    port: 8081
    address: "127.0.0.1"
```

13.3.5. Disabling HTTP Endpoints

If you do not want to expose endpoints over HTTP, you can set the management port to `-1`, as the following example shows:

Properties

```
management.server.port=-1
```

Yaml

```
management:
  server:
    port: -1
```

You can also achieve this by using the `management.endpoints.web.exposure.exclude` property, as the following example shows:

Properties

```
management.endpoints.web.exposure.exclude=*
```

Yaml

```
management:
  endpoints:
    web:
      exposure:
        exclude: "*"
```

13.4. Monitoring and Management over JMX

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. By default, this feature is not enabled. You can turn it on by setting the `spring.jmx.enabled` configuration property to `true`. Spring Boot exposes the most suitable `MBeanServer` as a bean with an ID of `mbeanServer`. Any of your beans that are annotated with Spring JMX annotations (`@ManagedResource`, `@ManagedAttribute`, or `@ManagedOperation`) are exposed to it.

If your platform provides a standard `MBeanServer`, Spring Boot uses that and defaults to the VM `MBeanServer`, if necessary. If all that fails, a new `MBeanServer` is created.

See the `JmxAutoConfiguration` class for more details.

By default, Spring Boot also exposes management endpoints as JMX MBeans under the `org.springframework.boot` domain. To take full control over endpoint registration in the JMX domain, consider registering your own `EndpointObjectNameFactory` implementation.

13.4.1. Customizing MBean Names

The name of the MBean is usually generated from the `id` of the endpoint. For example, the `health` endpoint is exposed as `org.springframework.boot:type=Endpoint,name=Health`.

If your application contains more than one Spring `ApplicationContext`, you may find that names clash. To solve this problem, you can set the `spring.jmx.unique-names` property to `true` so that MBean names are always unique.

You can also customize the JMX domain under which endpoints are exposed. The following settings

show an example of doing so in `application.properties`:

Properties

```
spring.jmx.unique-names=true
management.endpoints.jmx.domain=com.example.myapp
```

Yaml

```
spring:
  jmx:
    unique-names: true
management:
  endpoints:
    jmx:
      domain: "com.example.myapp"
```

13.4.2. Disabling JMX Endpoints

If you do not want to expose endpoints over JMX, you can set the `management.endpoints.jmx.exposure.exclude` property to `*`, as the following example shows:

Properties

```
management.endpoints.jmx.exposure.exclude=*
```

Yaml

```
management:
  endpoints:
    jmx:
      exposure:
        exclude: "*"
```

13.5. Observability

Observability is the ability to observe the internal state of a running system from the outside. It consists of the three pillars logging, metrics and traces.

For metrics and traces, Spring Boot uses Micrometer Observation. To create your own observations (which will lead to metrics and traces), you can inject an `ObservationRegistry`.

```
import io.micrometer.observation.Observation;
import io.micrometer.observation.ObservationRegistry;

import org.springframework.stereotype.Component;

@Component
public class MyCustomObservation {

    private final ObservationRegistry observationRegistry;

    public MyCustomObservation(ObservationRegistry observationRegistry) {
        this.observationRegistry = observationRegistry;
    }

    public void doSomething() {
        Observation.createNotStarted("doSomething", this.observationRegistry)
            .lowCardinalityKeyValue("locale", "en-US")
            .highCardinalityKeyValue("userId", "42")
            .observe(() -> {
                // Execute business logic here
            });
    }

}
```

NOTE	Low cardinality tags will be added to metrics and traces, while high cardinality tags will only be added to traces.

Beans of type `ObservationPredicate`, `GlobalObservationConvention` and `ObservationHandler` will be automatically registered on the `ObservationRegistry`. You can additionally register any number of `ObservationRegistryCustomizer` beans to further configure the registry.

For more details please see the Micrometer Observation documentation.

TIP	Observability for JDBC and R2DBC can be configured using separate projects. For JDBC, the Datasource Micrometer project provides a Spring Boot starter which automatically creates observations when JDBC operations are invoked. Read more about it in the reference documentation. For R2DBC, the Spring Boot Auto Configuration for R2DBC Observation creates observations for R2DBC query invocations.

The next sections will provide more details about logging, metrics and traces.

13.6. Loggers

Spring Boot Actuator includes the ability to view and configure the log levels of your application at runtime. You can view either the entire list or an individual logger's configuration, which is made up of both the explicitly configured logging level as well as the effective logging level given to it by

the logging framework. These levels can be one of:

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`
- `OFF`
- `null`

`null` indicates that there is no explicit configuration.

13.6.1. Configure a Logger

To configure a given logger, `POST` a partial entity to the resource's URI, as the following example shows:

```
{
    "configuredLevel": "DEBUG"
}
```

TIP To "reset" the specific level of the logger (and use the default configuration instead), you can pass a value of `null` as the `configuredLevel`.

13.7. Metrics

Spring Boot Actuator provides dependency management and auto-configuration for Micrometer, an application metrics facade that supports numerous monitoring systems, including:

- AppOptics
- Atlas
- Datadog
- Dynatrace
- Elastic
- Ganglia
- Graphite
- Humio
- Influx
- JMX

- KairosDB
- New Relic
- OpenTelemetry
- Prometheus
- SignalFx
- Simple (in-memory)
- Stackdriver
- StatsD
- Wavefront

TIP	To learn more about Micrometer's capabilities, see its reference documentation, in particular the concepts section.

13.7.1. Getting started

Spring Boot auto-configures a composite `MeterRegistry` and adds a registry to the composite for each of the supported implementations that it finds on the classpath. Having a dependency on `micrometer-registry-{system}` in your runtime classpath is enough for Spring Boot to configure the registry.

Most registries share common features. For instance, you can disable a particular registry even if the Micrometer registry implementation is on the classpath. The following example disables Datadog:

Properties

```
management.datadog.metrics.export.enabled=false
```

Yaml

```
management:
  datadog:
    metrics:
      export:
        enabled: false
```

You can also disable all registries unless stated otherwise by the registry-specific property, as the following example shows:

Properties

```
management.defaults.metrics.export.enabled=false
```

Yaml

```
management:
  defaults:
    metrics:
      export:
        enabled: false
```

Spring Boot also adds any auto-configured registries to the global static composite registry on the `Metrics` class, unless you explicitly tell it not to:

Properties

```
management.metrics.use-global-registry=false
```

Yaml

```
management:
  metrics:
    use-global-registry: false
```

You can register any number of `MeterRegistryCustomizer` beans to further configure the registry, such as applying common tags, before any meters are registered with the registry:

Java

```
import io.micrometer.core.instrument.MeterRegistry;

import org.springframework.boot.actuate.autoconfigure.metrics.MeterRegistryCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyMeterRegistryConfiguration {

    @Bean
    public MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {
        return (registry) -> registry.config().commonTags("region", "us-east-1");
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.MeterRegistry
import org.springframework.boot.actuate.autoconfigure.metrics.MeterRegistryCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyMeterRegistryConfiguration {

    @Bean
    fun metricsCommonTags(): MeterRegistryCustomizer<MeterRegistry> {
        return MeterRegistryCustomizer { registry ->
            registry.config().commonTags("region", "us-east-1")
        }
    }

}
```

You can apply customizations to particular registry implementations by being more specific about the generic type:

Java

```
import io.micrometer.core.instrument.Meter;
import io.micrometer.core.instrument.config.NamingConvention;
import io.micrometer.graphite.GraphiteMeterRegistry;

import org.springframework.boot.actuate.autoconfigure.metrics.MeterRegistryCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyMeterRegistryConfiguration {

    @Bean
    public MeterRegistryCustomizer<GraphiteMeterRegistry>
graphiteMetricsNamingConvention() {
        return (registry) -> registry.config().namingConvention(this::name);
    }

    private String name(String name, Meter.Type type, String baseUnit) {
        return ...
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.Meter
import io.micrometer.core.instrument.config.NamingConvention
import io.micrometer.graphite.GraphiteMeterRegistry
import org.springframework.boot.actuate.autoconfigure.metrics.MeterRegistryCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyMeterRegistryConfiguration {

    @Bean
    fun graphiteMetricsNamingConvention():
MeterRegistryCustomizer<GraphiteMeterRegistry> {
        return MeterRegistryCustomizer { registry: GraphiteMeterRegistry ->
            registry.config().namingConvention(this::name)
        }
    }

    private fun name(name: String, type: Meter.Type, baseUnit: String?): String {
        return  ...
    }

}
```

Spring Boot also configures built-in instrumentation that you can control through configuration or dedicated annotation markers.

13.7.2. Supported Monitoring Systems

This section briefly describes each of the supported monitoring systems.

AppOptics

By default, the AppOptics registry periodically pushes metrics to `api.appoptics.com/v1/measurements`. To export metrics to SaaS AppOptics, your API token must be provided:

Properties

```
management.appoptics.metrics.export.api-token=YOUR_TOKEN
```

Yaml

```
management:
  appoptics:
    metrics:
      export:
        api-token: "YOUR_TOKEN"
```

Atlas

By default, metrics are exported to Atlas running on your local machine. You can provide the location of the Atlas server:

Properties

```
management.atlas.metrics.export.uri=https://atlas.example.com:7101/api/v1/publish
```

Yaml

```
management:
  atlas:
    metrics:
      export:
        uri: "https://atlas.example.com:7101/api/v1/publish"
```

Datadog

A Datadog registry periodically pushes metrics to datadoghq. To export metrics to Datadog, you must provide your API key:

Properties

```
management.datadog.metrics.export.api-key=YOUR_KEY
```

Yaml

```
management:
  datadog:
    metrics:
      export:
        api-key: "YOUR_KEY"
```

If you additionally provide an application key (optional), then metadata such as meter descriptions, types, and base units will also be exported:

Properties

```
management.datadog.metrics.export.api-key=YOUR_API_KEY
management.datadog.metrics.export.application-key=YOUR_APPLICATION_KEY
```

Yaml

```
management:
  datadog:
    metrics:
      export:
        api-key: "YOUR_API_KEY"
        application-key: "YOUR_APPLICATION_KEY"
```

By default, metrics are sent to the Datadog US site (`api.datadoghq.com`). If your Datadog project is hosted on one of the other sites, or you need to send metrics through a proxy, configure the URI accordingly:

Properties

```
management.datadog.metrics.export.uri=https://api.datadoghq.eu
```

Yaml

```
management:
  datadog:
    metrics:
      export:
        uri: "https://api.datadoghq.eu"
```

You can also change the interval at which metrics are sent to Datadog:

Properties

```
management.datadog.metrics.export.step=30s
```

Yaml

```
management:
  datadog:
    metrics:
      export:
        step: "30s"
```

Dynatrace

Dynatrace offers two metrics ingest APIs, both of which are implemented for Micrometer. You can find the Dynatrace documentation on Micrometer metrics ingest here. Configuration properties in the `v1` namespace apply only when exporting to the Timeseries v1 API. Configuration properties in the `v2` namespace apply only when exporting to the Metrics v2 API. Note that this integration can export only to either the `v1` or `v2` version of the API at a time, with `v2` being preferred. If the `device-id` (required for v1 but not used in v2) is set in the `v1` namespace, metrics are exported to the `v1` endpoint. Otherwise, `v2` is assumed.

v2 API

You can use the v2 API in two ways.

Auto-configuration

Dynatrace auto-configuration is available for hosts that are monitored by the OneAgent or by the Dynatrace Operator for Kubernetes.

Local OneAgent: If a OneAgent is running on the host, metrics are automatically exported to the local OneAgent ingest endpoint. The ingest endpoint forwards the metrics to the Dynatrace backend.

Dynatrace Kubernetes Operator: When running in Kubernetes with the Dynatrace Operator installed, the registry will automatically pick up your endpoint URI and API token from the operator instead.

This is the default behavior and requires no special setup beyond a dependency on `io.micrometer:micrometer-registry-dynatrace`.

Manual configuration

If no auto-configuration is available, the endpoint of the Metrics v2 API and an API token are required. The API token must have the "Ingest metrics" (`metrics.ingest`) permission set. We recommend limiting the scope of the token to this one permission. You must ensure that the endpoint URI contains the path (for example, `/api/v2/metrics/ingest`):

The URL of the Metrics API v2 ingest endpoint is different according to your deployment option:

- SaaS: `https://{your-environment-id}.live.dynatrace.com/api/v2/metrics/ingest`
- Managed deployments: `https://{your-domain}/e/{your-environment-id}/api/v2/metrics/ingest`

The example below configures metrics export using the `example` environment id:

Properties

```
management.dynatrace.metrics.export.uri=https://example.live.dynatrace.com/api/v2/metr
ics/ingest
management.dynatrace.metrics.export.api-token=YOUR_TOKEN
```

Yaml

```
management:
  dynatrace:
    metrics:
      export:
        uri: "https://example.live.dynatrace.com/api/v2/metrics/ingest"
        api-token: "YOUR_TOKEN"
```

When using the Dynatrace v2 API, the following optional features are available (more details can be

found in the Dynatrace documentation):

- Metric key prefix: Sets a prefix that is prepended to all exported metric keys.
- Enrich with Dynatrace metadata: If a OneAgent or Dynatrace operator is running, enrich metrics with additional metadata (for example, about the host, process, or pod).
- Default dimensions: Specify key-value pairs that are added to all exported metrics. If tags with the same key are specified with Micrometer, they overwrite the default dimensions.
- Use Dynatrace Summary instruments: In some cases the Micrometer Dynatrace registry created metrics that were rejected. In Micrometer 1.9.x, this was fixed by introducing Dynatrace-specific summary instruments. Setting this toggle to `false` forces Micrometer to fall back to the behavior that was the default before 1.9.x. It should only be used when encountering problems while migrating from Micrometer 1.8.x to 1.9.x.

It is possible to not specify a URI and API token, as shown in the following example. In this scenario, the automatically configured endpoint is used:

Properties

```
management.dynatrace.metrics.export.v2.metric-key-prefix=your.key.prefix
management.dynatrace.metrics.export.v2.enrich-with-dynatrace-metadata=true
management.dynatrace.metrics.export.v2.default-dimensions.key1=value1
management.dynatrace.metrics.export.v2.default-dimensions.key2=value2
management.dynatrace.metrics.export.v2.use-dynatrace-summary-instruments=true
```

Yaml

```
management:
  dynatrace:
    metrics:
      export:
        # Specify uri and api-token here if not using the local OneAgent endpoint.
        v2:
          metric-key-prefix: "your.key.prefix"
          enrich-with-dynatrace-metadata: true
          default-dimensions:
            key1: "value1"
            key2: "value2"
          use-dynatrace-summary-instruments: true # (default: true)
```

v1 API (Legacy)

The Dynatrace v1 API metrics registry pushes metrics to the configured URI periodically by using the Timeseries v1 API. For backwards-compatibility with existing setups, when `device-id` is set (required for v1, but not used in v2), metrics are exported to the Timeseries v1 endpoint. To export metrics to Dynatrace, your API token, device ID, and URI must be provided:

Properties

```
management.dynatrace.metrics.export.uri=https://{your-environment-
id}.live.dynatrace.com
management.dynatrace.metrics.export.api-token=YOUR_TOKEN
management.dynatrace.metrics.export.v1.device-id=YOUR_DEVICE_ID
```

Yaml

```
management:
  dynatrace:
    metrics:
      export:
        uri: "https://{your-environment-id}.live.dynatrace.com"
        api-token: "YOUR_TOKEN"
        v1:
          device-id: "YOUR_DEVICE_ID"
```

For the v1 API, you must specify the base environment URI without a path, as the v1 endpoint path is added automatically.

Version-independent Settings

In addition to the API endpoint and token, you can also change the interval at which metrics are sent to Dynatrace. The default export interval is `60s`. The following example sets the export interval to 30 seconds:

Properties

```
management.dynatrace.metrics.export.step=30s
```

Yaml

```
management:
  dynatrace:
    metrics:
      export:
        step: "30s"
```

You can find more information on how to set up the Dynatrace exporter for Micrometer in the Micrometer documentation and the Dynatrace documentation.

Elastic

By default, metrics are exported to Elastic running on your local machine. You can provide the location of the Elastic server to use by using the following property:

Properties

```
management.elastic.metrics.export.host=https://elastic.example.com:8086
```

Yaml

```
management:
  elastic:
    metrics:
      export:
        host: "https://elastic.example.com:8086"
```

Ganglia

By default, metrics are exported to Ganglia running on your local machine. You can provide the Ganglia server host and port, as the following example shows:

Properties

```
management.ganglia.metrics.export.host=ganglia.example.com
management.ganglia.metrics.export.port=9649
```

Yaml

```
management:
  ganglia:
    metrics:
      export:
        host: "ganglia.example.com"
        port: 9649
```

Graphite

By default, metrics are exported to Graphite running on your local machine. You can provide the Graphite server host and port, as the following example shows:

Properties

```
management.graphite.metrics.export.host=graphite.example.com
management.graphite.metrics.export.port=9004
```

Yaml

```
management:
  graphite:
    metrics:
      export:
         host: "graphite.example.com"
         port: 9004
```

Micrometer provides a default `HierarchicalNameMapper` that governs how a dimensional meter ID is mapped to flat hierarchical names.

TIP

To take control over this behavior, define your `GraphiteMeterRegistry` and supply your own `HierarchicalNameMapper`. An auto-configured `GraphiteConfig` and `Clock` beans are provided unless you define your own:

Java

```
import io.micrometer.core.instrument.Clock;
import io.micrometer.core.instrument.Meter;
import io.micrometer.core.instrument.config.NamingConvention;
import io.micrometer.core.instrument.util.HierarchicalNameMapper;
import io.micrometer.graphite.GraphiteConfig;
import io.micrometer.graphite.GraphiteMeterRegistry;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyGraphiteConfiguration {

    @Bean
    public GraphiteMeterRegistry graphiteMeterRegistry(GraphiteConfig
config, Clock clock) {
        return new GraphiteMeterRegistry(config, clock,
this::toHierarchicalName);
    }

    private String toHierarchicalName(Meter.Id id, NamingConvention
convention) {
        return ...
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.Clock
import io.micrometer.core.instrument.Meter
import io.micrometer.core.instrument.config.NamingConvention
import io.micrometer.core.instrument.util.HierarchicalNameMapper
import io.micrometer.graphite.GraphiteConfig
import io.micrometer.graphite.GraphiteMeterRegistry
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyGraphiteConfiguration {

    @Bean
    fun graphiteMeterRegistry(config: GraphiteConfig, clock: Clock):
GraphiteMeterRegistry {
        return GraphiteMeterRegistry(config, clock,
this::toHierarchicalName)
    }
    private fun toHierarchicalName(id: Meter.Id, convention:
NamingConvention): String {
        return  ...
    }

}
```

Humio

By default, the Humio registry periodically pushes metrics to cloud.humio.com. To export metrics to SaaS Humio, you must provide your API token:

Properties

```
management.humio.metrics.export.api-token=YOUR_TOKEN
```

Yaml

```
management:
  humio:
    metrics:
      export:
        api-token: "YOUR_TOKEN"
```

You should also configure one or more tags to identify the data source to which metrics are pushed:

Properties

```
management.humio.metrics.export.tags.alpha=a
management.humio.metrics.export.tags.bravo=b
```

Yaml

```
management:
  humio:
    metrics:
      export:
        tags:
          alpha: "a"
          bravo: "b"
```

Influx

By default, metrics are exported to an Influx v1 instance running on your local machine with the default configuration. To export metrics to InfluxDB v2, configure the `org`, `bucket`, and authentication `token` for writing metrics. You can provide the location of the Influx server to use by using:

Properties

```
management.influx.metrics.export.uri=https://influx.example.com:8086
```

Yaml

```
management:
  influx:
    metrics:
      export:
        uri: "https://influx.example.com:8086"
```

JMX

Micrometer provides a hierarchical mapping to JMX, primarily as a cheap and portable way to view metrics locally. By default, metrics are exported to the `metrics` JMX domain. You can provide the domain to use by using:

Properties

```
management.jmx.metrics.export.domain=com.example.app.metrics
```

Yaml

```
management:
  jmx:
    metrics:
      export:
        domain: "com.example.app.metrics"
```

Micrometer provides a default `HierarchicalNameMapper` that governs how a dimensional meter ID is mapped to flat hierarchical names.

TIP

To take control over this behavior, define your `JmxMeterRegistry` and supply your own `HierarchicalNameMapper`. An auto-configured `JmxConfig` and `Clock` beans are provided unless you define your own:

Java

```
import io.micrometer.core.instrument.Clock;
import io.micrometer.core.instrument.Meter;
import io.micrometer.core.instrument.config.NamingConvention;
import io.micrometer.core.instrument.util.HierarchicalNameMapper;
import io.micrometer.jmx.JmxConfig;
import io.micrometer.jmx.JmxMeterRegistry;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyJmxConfiguration {

    @Bean
    public JmxMeterRegistry jmxMeterRegistry(JmxConfig config, Clock
clock) {
        return new JmxMeterRegistry(config, clock,
this::toHierarchicalName);
    }

    private String toHierarchicalName(Meter.Id id, NamingConvention
convention) {
        return ...
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.Clock
import io.micrometer.core.instrument.Meter
import io.micrometer.core.instrument.config.NamingConvention
import io.micrometer.core.instrument.util.HierarchicalNameMapper
import io.micrometer.jmx.JmxConfig
import io.micrometer.jmx.JmxMeterRegistry
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyJmxConfiguration {

    @Bean
    fun jmxMeterRegistry(config: JmxConfig, clock: Clock):
JmxMeterRegistry {
        return JmxMeterRegistry(config, clock, this::toHierarchicalName)
    }

    private fun toHierarchicalName(id: Meter.Id, convention:
NamingConvention): String {
        return  ...
    }

}
```

KairosDB

By default, metrics are exported to KairosDB running on your local machine. You can provide the location of the KairosDB server to use by using:

Properties

```
management.kairos.metrics.export.uri=https://kairosdb.example.com:8080/api/v1/datapoin
ts
```

Yaml

```
management:
  kairos:
    metrics:
      export:
        uri: "https://kairosdb.example.com:8080/api/v1/datapoints"
```

New Relic

A New Relic registry periodically pushes metrics to New Relic. To export metrics to New Relic, you must provide your API key and account ID:

Properties

```
management.newrelic.metrics.export.api-key=YOUR_KEY
management.newrelic.metrics.export.account-id=YOUR_ACCOUNT_ID
```

Yaml

```
management:
  newrelic:
    metrics:
      export:
        api-key: "YOUR_KEY"
        account-id: "YOUR_ACCOUNT_ID"
```

You can also change the interval at which metrics are sent to New Relic:

Properties

```
management.newrelic.metrics.export.step=30s
```

Yaml

```
management:
  newrelic:
    metrics:
      export:
        step: "30s"
```

By default, metrics are published through REST calls, but you can also use the Java Agent API if you have it on the classpath:

Properties

```
management.newrelic.metrics.export.client-provider-type=insights-agent
```

Yaml

```
management:
  newrelic:
    metrics:
      export:
        client-provider-type: "insights-agent"
```

Finally, you can take full control by defining your own `NewRelicClientProvider` bean.

OpenTelemetry

By default, metrics are exported to OpenTelemetry running on your local machine. You can provide the location of the OpenTelemetry metric endpoint to use by using:

Properties

```
management.otlp.metrics.export.url=https://otlp.example.com:4318/v1/metrics
```

Yaml

```
management:
  otlp:
    metrics:
      export:
        url: "https://otlp.example.com:4318/v1/metrics"
```

Prometheus

Prometheus expects to scrape or poll individual application instances for metrics. Spring Boot provides an actuator endpoint at `/actuator/prometheus` to present a Prometheus scrape with the appropriate format.

TIP	By default, the endpoint is not available and must be exposed. See exposing endpoints for more details.

The following example `scrape_config` adds to `prometheus.yml`:

```
scrape_configs:
  - job_name: "spring"
    metrics_path: "/actuator/prometheus"
    static_configs:
      - targets: ["HOST:PORT"]
```

Prometheus Exemplars are also supported. To enable this feature, a `SpanContextSupplier` bean should be present. If you use Micrometer Tracing, this will be auto-configured for you, but you can always create your own if you want. Please check the Prometheus Docs, since this feature needs to be explicitly enabled on Prometheus' side, and it is only supported using the OpenMetrics format.

For ephemeral or batch jobs that may not exist long enough to be scraped, you can use Prometheus Pushgateway support to expose the metrics to Prometheus. To enable Prometheus Pushgateway support, add the following dependency to your project:

```
<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_pushgateway</artifactId>
</dependency>
```

When the Prometheus Pushgateway dependency is present on the classpath and the `management.prometheus.metrics.export.pushgateway.enabled` property is set to `true`, a `PrometheusPushGatewayManager` bean is auto-configured. This manages the pushing of metrics to a Prometheus Pushgateway.

You can tune the `PrometheusPushGatewayManager` by using properties under `management.prometheus.metrics.export.pushgateway`. For advanced configuration, you can also provide your own `PrometheusPushGatewayManager` bean.

SignalFx

SignalFx registry periodically pushes metrics to SignalFx. To export metrics to SignalFx, you must provide your access token:

Properties

```
management.signalfx.metrics.export.access-token=YOUR_ACCESS_TOKEN
```

Yaml

```
management:
  signalfx:
    metrics:
      export:
        access-token: "YOUR_ACCESS_TOKEN"
```

You can also change the interval at which metrics are sent to SignalFx:

Properties

```
management.signalfx.metrics.export.step=30s
```

Yaml

```
management:
  signalfx:
    metrics:
      export:
        step: "30s"
```

Simple

Micrometer ships with a simple, in-memory backend that is automatically used as a fallback if no other registry is configured. This lets you see what metrics are collected in the metrics endpoint.

The in-memory backend disables itself as soon as you use any other available backend. You can also disable it explicitly:

Properties

```
management.simple.metrics.export.enabled=false
```

Yaml

```
management:
  simple:
    metrics:
      export:
        enabled: false
```

Stackdriver

The Stackdriver registry periodically pushes metrics to Stackdriver. To export metrics to SaaS Stackdriver, you must provide your Google Cloud project ID:

Properties

```
management.stackdriver.metrics.export.project-id=my-project
```

Yaml

```
management:
  stackdriver:
    metrics:
      export:
        project-id: "my-project"
```

You can also change the interval at which metrics are sent to Stackdriver:

Properties

```
management.stackdriver.metrics.export.step=30s
```

Yaml

```
management:
  stackdriver:
    metrics:
      export:
        step: "30s"
```

StatsD

The StatsD registry eagerly pushes metrics over UDP to a StatsD agent. By default, metrics are exported to a StatsD agent running on your local machine. You can provide the StatsD agent host,

port, and protocol to use by using:

Properties

```
management.statsd.metrics.export.host=statsd.example.com
management.statsd.metrics.export.port=9125
management.statsd.metrics.export.protocol=udp
```

Yaml

```
management:
  statsd:
    metrics:
      export:
        host: "statsd.example.com"
        port: 9125
        protocol: "udp"
```

You can also change the StatsD line protocol to use (it defaults to Datadog):

Properties

```
management.statsd.metrics.export.flavor=etsy
```

Yaml

```
management:
  statsd:
    metrics:
      export:
        flavor: "etsy"
```

Wavefront

The Wavefront registry periodically pushes metrics to Wavefront. If you are exporting metrics to Wavefront directly, you must provide your API token:

Properties

```
management.wavefront.api-token=YOUR_API_TOKEN
```

Yaml

```
management:
  wavefront:
    api-token: "YOUR_API_TOKEN"
```

Alternatively, you can use a Wavefront sidecar or an internal proxy in your environment to

forward metrics data to the Wavefront API host:

Properties

```
management.wavefront.uri=proxy://localhost:2878
```

Yaml

```
management:
  wavefront:
    uri: "proxy://localhost:2878"
```

NOTE If you publish metrics to a Wavefront proxy (as described in the Wavefront documentation), the host must be in the `proxy://HOST:PORT` format.

You can also change the interval at which metrics are sent to Wavefront:

Properties

```
management.wavefront.metrics.export.step=30s
```

Yaml

```
management:
  wavefront:
    metrics:
      export:
        step: "30s"
```

13.7.3. Supported Metrics and Meters

Spring Boot provides automatic meter registration for a wide variety of technologies. In most situations, the defaults provide sensible metrics that can be published to any of the supported monitoring systems.

JVM Metrics

Auto-configuration enables JVM Metrics by using core Micrometer classes. JVM metrics are published under the `jvm.` meter name.

The following JVM metrics are provided:

- Various memory and buffer pool details
- Statistics related to garbage collection
- Thread utilization
- The number of classes loaded and unloaded

- JVM version information
- JIT compilation time

System Metrics

Auto-configuration enables system metrics by using core Micrometer classes. System metrics are published under the `system.`, `process.`, and `disk.` meter names.

The following system metrics are provided:

- CPU metrics
- File descriptor metrics
- Uptime metrics (both the amount of time the application has been running and a fixed gauge of the absolute start time)
- Disk space available

Application Startup Metrics

Auto-configuration exposes application startup time metrics:

- `application.started.time`: time taken to start the application.
- `application.ready.time`: time taken for the application to be ready to service requests.

Metrics are tagged by the fully qualified name of the application class.

Logger Metrics

Auto-configuration enables the event metrics for both Logback and Log4J2. The details are published under the `log4j2.events.` or `logback.events.` meter names.

Task Execution and Scheduling Metrics

Auto-configuration enables the instrumentation of all available `ThreadPoolTaskExecutor` and `ThreadPoolTaskScheduler` beans, as long as the underling `ThreadPoolExecutor` is available. Metrics are tagged by the name of the executor, which is derived from the bean name.

Spring MVC Metrics

Auto-configuration enables the instrumentation of all requests handled by Spring MVC controllers and functional handlers. By default, metrics are generated with the name, `http.server.requests`. You can customize the name by setting the `management.observations.http.server.requests.name` property.

See the Spring Framework reference documentation for more information on produced observations.

To add to the default tags, provide a `@Bean` that extends `DefaultServerRequestObservationConvention` from the `org.springframework.http.server.observation` package. To replace the default tags, provide a `@Bean` that implements `ServerRequestObservationConvention`.

TIP	In some cases, exceptions handled in web controllers are not recorded as request metrics tags. Applications can opt in and record exceptions by setting handled exceptions as request attributes.

By default, all requests are handled. To customize the filter, provide a `@Bean` that implements `FilterRegistrationBean<WebMvcMetricsFilter>`.

Spring WebFlux Metrics

Auto-configuration enables the instrumentation of all requests handled by Spring WebFlux controllers and functional handlers. By default, metrics are generated with the name, `http.server.requests`. You can customize the name by setting the `management.observations.http.server.requests.name` property.

See the Spring Framework reference documentation for more information on produced observations.

To add to the default tags, provide a `@Bean` that extends `DefaultServerRequestObservationConvention` from the `org.springframework.http.server.reactive.observation` package. To replace the default tags, provide a `@Bean` that implements `ServerRequestObservationConvention`.

TIP	In some cases, exceptions handled in controllers and handler functions are not recorded as request metrics tags. Applications can opt in and record exceptions by setting handled exceptions as request attributes.

Jersey Server Metrics

Auto-configuration enables the instrumentation of all requests handled by the Jersey JAX-RS implementation. By default, metrics are generated with the name, `http.server.requests`. You can customize the name by setting the `management.observations.http.server.requests.name` property.

By default, Jersey server metrics are tagged with the following information:

Tag	Description
`exception`	The simple class name of any exception that was thrown while handling the request.
`method`	The request's method (for example, `GET` or `POST`)
`outcome`	The request's outcome, based on the status code of the response. 1xx is `INFORMATIONAL`, 2xx is `SUCCESS`, 3xx is `REDIRECTION`, 4xx is `CLIENT_ERROR`, and 5xx is `SERVER_ERROR`
`status`	The response's HTTP status code (for example, `200` or `500`)
`uri`	The request's URI template prior to variable substitution, if possible (for example, `/api/person/{id}`)

To customize the tags, provide a `@Bean` that implements `JerseyTagsProvider`.

HTTP Client Metrics

Spring Boot Actuator manages the instrumentation of both `RestTemplate` and `WebClient`. For that, you have to inject the auto-configured builder and use it to create instances:

- `RestTemplateBuilder` for `RestTemplate`
- `WebClient.Builder` for `WebClient`

You can also manually apply the customizers responsible for this instrumentation, namely `ObservationRestTemplateCustomizer` and `ObservationWebClientCustomizer`.

By default, metrics are generated with the name, `http.client.requests`. You can customize the name by setting the `management.observations.http.client.requests.name` property.

See the Spring Framework reference documentation for more information on produced observations.

To customize the tags when using `RestTemplate`, provide a `@Bean` that implements `ClientRequestObservationConvention` from the `org.springframework.http.client.observation` package. To customize the tags when using `WebClient`, provide a `@Bean` that implements `ClientRequestObservationConvention` from the `org.springframework.web.reactive.function.client` package.

Tomcat Metrics

Auto-configuration enables the instrumentation of Tomcat only when an `MBeanRegistry` is enabled. By default, the `MBeanRegistry` is disabled, but you can enable it by setting `server.tomcat.mbeanregistry.enabled` to `true`.

Tomcat metrics are published under the `tomcat.` meter name.

Cache Metrics

Auto-configuration enables the instrumentation of all available `Cache` instances on startup, with metrics prefixed with `cache`. Cache instrumentation is standardized for a basic set of metrics. Additional, cache-specific metrics are also available.

The following cache libraries are supported:

- Cache2k
- Caffeine
- Hazelcast
- Any compliant JCache (JSR-107) implementation
- Redis

Metrics are tagged by the name of the cache and by the name of the `CacheManager`, which is derived from the bean name.

NOTE	Only caches that are configured on startup are bound to the registry. For caches not defined in the cache's configuration, such as caches created on the fly or programmatically after the startup phase, an explicit registration is required. A `CacheMetricsRegistrar` bean is made available to make that process easier.

Spring Batch Metrics

See the Spring Batch reference documentation.

Spring GraphQL Metrics

See the Spring GraphQL reference documentation.

DataSource Metrics

Auto-configuration enables the instrumentation of all available `DataSource` objects with metrics prefixed with `jdbc.connections`. Data source instrumentation results in gauges that represent the currently active, idle, maximum allowed, and minimum allowed connections in the pool.

Metrics are also tagged by the name of the `DataSource` computed based on the bean name.

TIP	By default, Spring Boot provides metadata for all supported data sources. You can add additional `DataSourcePoolMetadataProvider` beans if your favorite data source is not supported. See `DataSourcePoolMetadataProvidersConfiguration` for examples.

Also, Hikari-specific metrics are exposed with a `hikaricp` prefix. Each metric is tagged by the name of the pool (you can control it with `spring.datasource.name`).

Hibernate Metrics

If `org.hibernate.orm:hibernate-micrometer` is on the classpath, all available Hibernate `EntityManagerFactory` instances that have statistics enabled are instrumented with a metric named `hibernate`.

Metrics are also tagged by the name of the `EntityManagerFactory`, which is derived from the bean name.

To enable statistics, the standard JPA property `hibernate.generate_statistics` must be set to `true`. You can enable that on the auto-configured `EntityManagerFactory`:

Properties

```
spring.jpa.properties[hibernate.generate_statistics]=true
```

Yaml

```
spring:
  jpa:
    properties:
      "[hibernate.generate_statistics]": true
```

Spring Data Repository Metrics

Auto-configuration enables the instrumentation of all Spring Data `Repository` method invocations. By default, metrics are generated with the name, `spring.data.repository.invocations`. You can customize the name by setting the `management.metrics.data.repository.metric-name` property.

The `@Timed` annotation from the `io.micrometer.core.annotation` package is supported on `Repository` interfaces and methods. If you do not want to record metrics for all `Repository` invocations, you can set `management.metrics.data.repository.autotime.enabled` to `false` and exclusively use `@Timed` annotations instead.

NOTE	A `@Timed` annotation with `longTask = true` enables a long task timer for the method. Long task timers require a separate metric name and can be stacked with a short task timer.

By default, repository invocation related metrics are tagged with the following information:

Tag	Description
`repository`	The simple class name of the source `Repository`.
`method`	The name of the `Repository` method that was invoked.
`state`	The result state (`SUCCESS`, `ERROR`, `CANCELED`, or `RUNNING`).
`exception`	The simple class name of any exception that was thrown from the invocation.

To replace the default tags, provide a `@Bean` that implements `RepositoryTagsProvider`.

RabbitMQ Metrics

Auto-configuration enables the instrumentation of all available RabbitMQ connection factories with a metric named `rabbitmq`.

Spring Integration Metrics

Spring Integration automatically provides Micrometer support whenever a `MeterRegistry` bean is available. Metrics are published under the `spring.integration.` meter name.

Kafka Metrics

Auto-configuration registers a `MicrometerConsumerListener` and `MicrometerProducerListener` for the auto-configured consumer factory and producer factory, respectively. It also registers a `KafkaStreamsMicrometerListener` for `StreamsBuilderFactoryBean`. For more detail, see the Micrometer Native Metrics section of the Spring Kafka documentation.

MongoDB Metrics

This section briefly describes the available metrics for MongoDB.

MongoDB Command Metrics

Auto-configuration registers a `MongoMetricsCommandListener` with the auto-configured `MongoClient`.

A timer metric named `mongodb.driver.commands` is created for each command issued to the underlying MongoDB driver. Each metric is tagged with the following information by default:

Tag	Description
`command`	The name of the command issued.
`cluster.id`	The identifier of the cluster to which the command was sent.
`server.address`	The address of the server to which the command was sent.
`status`	The outcome of the command (`SUCCESS` or `FAILED`).

To replace the default metric tags, define a `MongoCommandTagsProvider` bean, as the following example shows:

Java

```
import io.micrometer.core.instrument.binder.mongodb.MongoCommandTagsProvider;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyCommandTagsProviderConfiguration {

    @Bean
    public MongoCommandTagsProvider customCommandTagsProvider() {
        return new CustomCommandTagsProvider();
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.binder.mongodb.MongoCommandTagsProvider
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyCommandTagsProviderConfiguration {

    @Bean
    fun customCommandTagsProvider(): MongoCommandTagsProvider? {
        return CustomCommandTagsProvider()
    }

}
```

To disable the auto-configured command metrics, set the following property:

Properties

```
management.metrics.mongo.command.enabled=false
```

Yaml

```
management:
  metrics:
    mongo:
      command:
        enabled: false
```

MongoDB Connection Pool Metrics

Auto-configuration registers a `MongoMetricsConnectionPoolListener` with the auto-configured `MongoClient`.

The following gauge metrics are created for the connection pool:

- `mongodb.driver.pool.size` reports the current size of the connection pool, including idle and and in-use members.
- `mongodb.driver.pool.checkedout` reports the count of connections that are currently in use.
- `mongodb.driver.pool.waitqueuesize` reports the current size of the wait queue for a connection from the pool.

Each metric is tagged with the following information by default:

Tag	Description
`cluster.id`	The identifier of the cluster to which the connection pool corresponds.

Tag	Description
`server.address`	The address of the server to which the connection pool corresponds.

To replace the default metric tags, define a `MongoConnectionPoolTagsProvider` bean:

Java

```
import io.micrometer.core.instrument.binder.mongodb.MongoConnectionPoolTagsProvider;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyConnectionPoolTagsProviderConfiguration {

    @Bean
    public MongoConnectionPoolTagsProvider customConnectionPoolTagsProvider() {
        return new CustomConnectionPoolTagsProvider();
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.binder.mongodb.MongoConnectionPoolTagsProvider
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyConnectionPoolTagsProviderConfiguration {

    @Bean
    fun customConnectionPoolTagsProvider(): MongoConnectionPoolTagsProvider {
        return CustomConnectionPoolTagsProvider()
    }

}
```

To disable the auto-configured connection pool metrics, set the following property:

Properties

```
management.metrics.mongo.connectionpool.enabled=false
```

Yaml

```
management:
  metrics:
    mongo:
      connectionpool:
        enabled: false
```

Jetty Metrics

Auto-configuration binds metrics for Jetty's `ThreadPool` by using Micrometer's `JettyServerThreadPoolMetrics`. Metrics for Jetty's `Connector` instances are bound by using Micrometer's `JettyConnectionMetrics` and, when `server.ssl.enabled` is set to `true`, Micrometer's `JettySslHandshakeMetrics`.

@Timed Annotation Support

To use `@Timed` where it is not directly supported by Spring Boot, refer to the Micrometer documentation.

Redis Metrics

Auto-configuration registers a `MicrometerCommandLatencyRecorder` for the auto-configured `LettuceConnectionFactory`. For more detail, see the Micrometer Metrics section of the Lettuce documentation.

13.7.4. Registering Custom Metrics

To register custom metrics, inject `MeterRegistry` into your component:

Java

```
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.Tags;

import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final Dictionary dictionary;

    public MyBean(MeterRegistry registry) {
        this.dictionary = Dictionary.load();
        registry.gauge("dictionary.size", Tags.empty(),
this.dictionary.getWords().size());
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.MeterRegistry
import io.micrometer.core.instrument.Tags
import org.springframework.stereotype.Component

@Component
class MyBean(registry: MeterRegistry) {

    private val dictionary: Dictionary

    init {
        dictionary = Dictionary.load()
        registry.gauge("dictionary.size", Tags.empty(), dictionary.words.size)
    }

}
```

If your metrics depend on other beans, we recommend that you use a `MeterBinder` to register them:

Java

```
import io.micrometer.core.instrument.Gauge;
import io.micrometer.core.instrument.binder.MeterBinder;

import org.springframework.context.annotation.Bean;

public class MyMeterBinderConfiguration {

    @Bean
    public MeterBinder queueSize(Queue queue) {
        return (registry) -> Gauge.builder("queueSize",
queue::size).register(registry);
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.Gauge
import io.micrometer.core.instrument.binder.MeterBinder
import org.springframework.context.annotation.Bean

class MyMeterBinderConfiguration {

    @Bean
    fun queueSize(queue: Queue): MeterBinder {
        return MeterBinder { registry ->
            Gauge.builder("queueSize", queue::size).register(registry)
        }
    }

}
```

Using a `MeterBinder` ensures that the correct dependency relationships are set up and that the bean is available when the metric's value is retrieved. A `MeterBinder` implementation can also be useful if you find that you repeatedly instrument a suite of metrics across components or applications.

NOTE By default, metrics from all `MeterBinder` beans are automatically bound to the Spring-managed `MeterRegistry`.

13.7.5. Customizing Individual Metrics

If you need to apply customizations to specific `Meter` instances, you can use the `io.micrometer.core.instrument.config.MeterFilter` interface.

For example, if you want to rename the `mytag.region` tag to `mytag.area` for all meter IDs beginning with `com.example`, you can do the following:

Java

```
import io.micrometer.core.instrument.config.MeterFilter;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyMetricsFilterConfiguration {

    @Bean
    public MeterFilter renameRegionTagMeterFilter() {
        return MeterFilter.renameTag("com.example", "mytag.region", "mytag.area");
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.config.MeterFilter
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyMetricsFilterConfiguration {

    @Bean
    fun renameRegionTagMeterFilter(): MeterFilter {
        return MeterFilter.renameTag("com.example", "mytag.region", "mytag.area")
    }

}
```

NOTE By default, all `MeterFilter` beans are automatically bound to the Spring-managed `MeterRegistry`. Make sure to register your metrics by using the Spring-managed `MeterRegistry` and not any of the static methods on `Metrics`. These use the global registry that is not Spring-managed.

Common Tags

Common tags are generally used for dimensional drill-down on the operating environment, such as host, instance, region, stack, and others. Commons tags are applied to all meters and can be configured, as the following example shows:

Properties

```
management.metrics.tags.region=us-east-1
management.metrics.tags.stack=prod
```

Yaml

```
management:
  metrics:
    tags:
      region: "us-east-1"
      stack: "prod"
```

The preceding example adds `region` and `stack` tags to all meters with a value of `us-east-1` and `prod`, respectively.

NOTE The order of common tags is important if you use Graphite. As the order of common tags cannot be guaranteed by using this approach, Graphite users are advised to define a custom `MeterFilter` instead.

Per-meter Properties

In addition to `MeterFilter` beans, you can apply a limited set of customization on a per-meter basis using properties. Per-meter customizations are applied, using Spring Boot's `PropertiesMeterFilter`, to any meter IDs that start with the given name. The following example filters out any meters that have an ID starting with `example.remote`.

Properties

```
management.metrics.enable.example.remote=false
```

Yaml

```
management:
  metrics:
    enable:
      example:
        remote: false
```

The following properties allow per-meter customization:

Table 9. Per-meter customizations

Property	Description
`management.metrics.enable`	Whether to accept meters with certain IDs. Meters that are not accepted are filtered from the `MeterRegistry`.
`management.metrics.distribution.percentiles-histogram`	Whether to publish a histogram suitable for computing aggregable (across dimension) percentile approximations.
`management.metrics.distribution.minimum-expected-value`, `management.metrics.distribution.maximum-expected-value`	Publish fewer histogram buckets by clamping the range of expected values.
`management.metrics.distribution.percentiles`	Publish percentile values computed in your application
`management.metrics.distribution.expiry`, `management.metrics.distribution.buffer-length`	Give greater weight to recent samples by accumulating them in ring buffers which rotate after a configurable expiry, with a configurable buffer length.
`management.metrics.distribution.slo`	Publish a cumulative histogram with buckets defined by your service-level objectives.

For more details on the concepts behind `percentiles-histogram`, `percentiles`, and `slo`, see the "Histograms and percentiles" section of the Micrometer documentation.

13.7.6. Metrics Endpoint

Spring Boot provides a `metrics` endpoint that you can use diagnostically to examine the metrics collected by an application. The endpoint is not available by default and must be exposed. See exposing endpoints for more details.

Navigating to `/actuator/metrics` displays a list of available meter names. You can drill down to view information about a particular meter by providing its name as a selector — for example, `/actuator/metrics/jvm.memory.max`.

TIP The name you use here should match the name used in the code, not the name after it has been naming-convention normalized for a monitoring system to which it is shipped. In other words, if `jvm.memory.max` appears as `jvm_memory_max` in Prometheus because of its snake case naming convention, you should still use `jvm.memory.max` as the selector when inspecting the meter in the `metrics` endpoint.

You can also add any number of `tag=KEY:VALUE` query parameters to the end of the URL to dimensionally drill down on a meter — for example, `/actuator/metrics/jvm.memory.max?tag=area:nonheap`.

TIP The reported measurements are the *sum* of the statistics of all meters that match the meter name and any tags that have been applied. In the preceding example, the returned `Value` statistic is the sum of the maximum memory footprints of the “Code Cache”, “Compressed Class Space”, and “Metaspace” areas of the heap. If you wanted to see only the maximum size for the “Metaspace”, you could add an additional `tag=id:Metaspace` — that is, `/actuator/metrics/jvm.memory.max?tag=area:nonheap&tag=id:Metaspace`.

13.7.7. Integration with Micrometer Observation

A `DefaultMeterObservationHandler` is automatically registered on the `ObservationRegistry`, which creates metrics for every completed observation.

13.8. Tracing

Spring Boot Actuator provides dependency management and auto-configuration for Micrometer Tracing, a facade for popular tracer libraries.

TIP To learn more about Micrometer Tracing capabilities, see its reference documentation.

13.8.1. Supported Tracers

Spring Boot ships auto-configuration for the following tracers:

- OpenTelemetry with Zipkin, Wavefront, or OTLP
- OpenZipkin Brave with Zipkin or Wavefront

13.8.2. Getting Started

We need an example application that we can use to get started with tracing. For our purposes, the simple "Hello World!" web application that's covered in the "Developing Your First Spring Boot Application" section will suffice. We're going to use the OpenTelemetry tracer with Zipkin as trace backend.

To recap, our main application code looks like this:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class MyApplication {

    private static final Log logger = LogFactory.getLog(MyApplication.class);

    @RequestMapping("/")
    String home() {
        logger.info("home() has been called");
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

NOTE	There's an added logger statement in the `home()` method, which will be important later.

Now we have to add the following dependencies:

- `org.springframework.boot:spring-boot-starter-actuator`
- `io.micrometer:micrometer-tracing-bridge-otel` - bridges the Micrometer Observation API to OpenTelemetry.
- `io.opentelemetry:opentelemetry-exporter-zipkin` - reports traces to Zipkin.

Add the following application properties:

Properties

```
management.tracing.sampling.probability=1.0
```

Yaml

```
management:
  tracing:
    sampling:
      probability: 1.0
```

By default, Spring Boot samples only 10% of requests to prevent overwhelming the trace backend. This property switches it to 100% so that every request is sent to the trace backend.

To collect and visualize the traces, we need a running trace backend. We use Zipkin as our trace backend here. The Zipkin Quickstart guide provides instructions how to start Zipkin locally.

After Zipkin is running, you can start your application.

If you open a web browser to `localhost:8080`, you should see the following output:

```
Hello World!
```

Behind the scenes, an observation has been created for the HTTP request, which in turn gets bridged to OpenTelemetry, which reports a new trace to Zipkin.

Now open the Zipkin UI at `localhost:9411` and press the "Run Query" button to list all collected traces. You should see one trace. Press the "Show" button to see the details of that trace.

TIP	You can include the current trace and span id in the logs by setting the `logging.pattern.level` property to `%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}]`

13.8.3. Propagating Traces

To automatically propagate traces over the network, use the auto-configured `RestTemplateBuilder` or `WebClient.Builder` to construct the client.

WARNING	If you create the `WebClient` or the `RestTemplate` without using the auto-configured builders, automatic trace propagation won't work!

13.8.4. Tracer Implementations

As Micrometer Tracer supports multiple tracer implementations, there are multiple dependency combinations possible with Spring Boot.

All tracer implementations need the `org.springframework.boot:spring-boot-starter-actuator` dependency.

OpenTelemetry With Zipkin

Tracing with OpenTelemetry and reporting to Zipkin requires the following dependencies:

- `io.micrometer:micrometer-tracing-bridge-otel` - bridges the Micrometer Observation API to OpenTelemetry.
- `io.opentelemetry:opentelemetry-exporter-zipkin` - reports traces to Zipkin.

Use the `management.zipkin.tracing.*` configuration properties to configure reporting to Zipkin.

OpenTelemetry With Wavefront

Tracing with OpenTelemetry and reporting to Wavefront requires the following dependencies:

- `io.micrometer:micrometer-tracing-bridge-otel` - bridges the Micrometer Observation API to OpenTelemetry.
- `io.micrometer:micrometer-tracing-reporter-wavefront` - reports traces to Wavefront.

Use the `management.wavefront.*` configuration properties to configure reporting to Wavefront.

OpenTelemetry With OTLP

Tracing with OpenTelemetry and reporting using OTLP requires the following dependencies:

- `io.micrometer:micrometer-tracing-bridge-otel` - bridges the Micrometer Observation API to OpenTelemetry.
- `io.opentelemetry:opentelemetry-exporter-otlp` - reports traces to a collector that can accept OTLP.

Use the `management.otlp.tracing.*` configuration properties to configure reporting using OTLP.

OpenZipkin Brave With Zipkin

Tracing with OpenZipkin Brave and reporting to Zipkin requires the following dependencies:

- `io.micrometer:micrometer-tracing-bridge-brave` - bridges the Micrometer Observation API to Brave.
- `io.zipkin.reporter2:zipkin-reporter-brave` - reports traces to Zipkin.

NOTE	If your project doesn't use Spring MVC or Spring WebFlux, the `io.zipkin.reporter2:zipkin-sender-urlconnection` dependency is needed, too.

Use the `management.zipkin.tracing.*` configuration properties to configure reporting to Zipkin.

OpenZipkin Brave With Wavefront

Tracing with OpenZipkin Brave and reporting to Wavefront requires the following dependencies:

- `io.micrometer:micrometer-tracing-bridge-brave` - bridges the Micrometer Observation API to Brave.

- `io.micrometer:micrometer-tracing-reporter-wavefront` - reports traces to Wavefront.

Use the `management.wavefront.*` configuration properties to configure reporting to Wavefront.

13.8.5. Integration with Micrometer Observation

A `TracingAwareMeterObservationHandler` is automatically registered on the `ObservationRegistry`, which creates spans for every completed observation.

13.8.6. Creating Custom Spans

You can create your own spans by starting an observation. For this, inject `ObservationRegistry` into your component:

```
import io.micrometer.observation.Observation;
import io.micrometer.observation.ObservationRegistry;

import org.springframework.stereotype.Component;

@Component
class CustomObservation {

    private final ObservationRegistry observationRegistry;

    CustomObservation(ObservationRegistry observationRegistry) {
        this.observationRegistry = observationRegistry;
    }

    void someOperation() {
        Observation observation = Observation.createNotStarted("some-operation",
this.observationRegistry);
        observation.lowCardinalityKeyValue("some-tag", "some-value");
        observation.observe(() -> {
            // Business logic ...
        });
    }

}
```

This will create an observation named "some-operation" with the tag "some-tag=some-value".

TIP If you want to create a span without creating a metric, you need to use the lower-level `Tracer` API from Micrometer.

13.8.7. Baggage

You can create baggage with the `Tracer` API:

```
import io.micrometer.tracing.BaggageInScope;
import io.micrometer.tracing.Tracer;

import org.springframework.stereotype.Component;

@Component
class CreatingBaggage {

    private final Tracer tracer;

    CreatingBaggage(Tracer tracer) {
        this.tracer = tracer;
    }

    void doSomething() {
        try (BaggageInScope scope = this.tracer.createBaggageInScope("baggage1",
"value1")) {
            // Business logic
        }
    }

}
```

This example creates baggage named `baggage1` with the value `value1`. The baggage is automatically propagated over the network if you're using W3C propagation. If you're using B3 propagation, baggage is not automatically propagated. To manually propagate baggage over the network, use the `management.tracing.baggage.remote-fields` configuration property (this works for W3C, too). For the example above, setting this property to `baggage1` results in an HTTP header `baggage1: value1`.

If you want to propagate the baggage to the MDC, use the `management.tracing.baggage.correlation.fields` configuration property. For the example above, setting this property to `baggage1` results in an MDC entry named `baggage1`.

13.8.8. Tests

Tracing is not auto-configured when using `@SpringBootTest`. See the testing section for more details.

13.9. Auditing

Once Spring Security is in play, Spring Boot Actuator has a flexible audit framework that publishes events (by default, "authentication success", "failure" and "access denied" exceptions). This feature can be very useful for reporting and for implementing a lock-out policy based on authentication failures.

You can enable auditing by providing a bean of type `AuditEventRepository` in your application's configuration. For convenience, Spring Boot offers an `InMemoryAuditEventRepository`. `InMemoryAuditEventRepository` has limited capabilities, and we recommend using it only for development environments. For production environments, consider creating your own alternative

`AuditEventRepository` implementation.

13.9.1. Custom Auditing

To customize published security events, you can provide your own implementations of `AbstractAuthenticationAuditListener` and `AbstractAuthorizationAuditListener`.

You can also use the audit services for your own business events. To do so, either inject the `AuditEventRepository` bean into your own components and use that directly or publish an `AuditApplicationEvent` with the Spring `ApplicationEventPublisher` (by implementing `ApplicationEventPublisherAware`).

13.10. Recording HTTP Exchanges

You can enable recording of HTTP exchanges by providing a bean of type `HttpExchangeRepository` in your application's configuration. For convenience, Spring Boot offers `InMemoryHttpExchangeRepository`, which, by default, stores the last 100 request-response exchanges. `InMemoryHttpExchangeRepository` is limited compared to tracing solutions, and we recommend using it only for development environments. For production environments, we recommend using a production-ready tracing or observability solution, such as Zipkin or OpenTelemetry. Alternatively, you can create your own `HttpExchangeRepository`.

You can use the `httpexchanges` endpoint to obtain information about the request-response exchanges that are stored in the `HttpExchangeRepository`.

13.10.1. Custom HTTP Exchange Recording

To customize the items that are included in each recorded exchange, use the `management.httpexchanges.recording.include` configuration property.

To disable recoding entirely, set `management.httpexchanges.recording.enabled` to `false`.

13.11. Process Monitoring

In the `spring-boot` module, you can find two classes to create files that are often useful for process monitoring:

- `ApplicationPidFileWriter` creates a file that contains the application PID (by default, in the application directory with a file name of `application.pid`).
- `WebServerPortFileWriter` creates a file (or files) that contain the ports of the running web server (by default, in the application directory with a file name of `application.port`).

By default, these writers are not activated, but you can enable them:

- By Extending Configuration
- Programmatically Enabling Process Monitoring

13.11.1. Extending Configuration

In the `META-INF/spring.factories` file, you can activate the listener (or listeners) that writes a PID file:

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.context.ApplicationPidFileWriter,\
org.springframework.boot.web.context.WebServerPortFileWriter
```

13.11.2. Programmatically Enabling Process Monitoring

You can also activate a listener by invoking the `SpringApplication.addListeners(…)` method and passing the appropriate `Writer` object. This method also lets you customize the file name and path in the `Writer` constructor.

13.12. Cloud Foundry Support

Spring Boot's actuator module includes additional support that is activated when you deploy to a compatible Cloud Foundry instance. The `/cloudfoundryapplication` path provides an alternative secured route to all `@Endpoint` beans.

The extended support lets Cloud Foundry management UIs (such as the web application that you can use to view deployed applications) be augmented with Spring Boot actuator information. For example, an application status page can include full health information instead of the typical "running" or "stopped" status.

NOTE	The `/cloudfoundryapplication` path is not directly accessible to regular users. To use the endpoint, you must pass a valid UAA token with the request.

13.12.1. Disabling Extended Cloud Foundry Actuator Support

If you want to fully disable the `/cloudfoundryapplication` endpoints, you can add the following setting to your `application.properties` file:

Properties

```
management.cloudfoundry.enabled=false
```

Yaml

```
management:
  cloudfoundry:
    enabled: false
```

13.12.2. Cloud Foundry Self-signed Certificates

By default, the security verification for `/cloudfoundryapplication` endpoints makes SSL calls to various Cloud Foundry services. If your Cloud Foundry UAA or Cloud Controller services use self-signed certificates, you need to set the following property:

Properties

```
management.cloudfoundry.skip-ssl-validation=true
```

Yaml

```
management:
  cloudfoundry:
    skip-ssl-validation: true
```

13.12.3. Custom Context Path

If the server's context-path has been configured to anything other than `/`, the Cloud Foundry endpoints are not available at the root of the application. For example, if `server.servlet.context-path=/app`, Cloud Foundry endpoints are available at `/app/cloudfoundryapplication/*`.

If you expect the Cloud Foundry endpoints to always be available at `/cloudfoundryapplication/*`, regardless of the server's context-path, you need to explicitly configure that in your application. The configuration differs, depending on the web server in use. For Tomcat, you can add the following configuration:

Java

```
import java.io.IOException;
import java.util.Collections;

import jakarta.servlet.GenericServlet;
import jakarta.servlet.Servlet;
import jakarta.servlet.ServletContainerInitializer;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import org.apache.catalina.Host;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.startup.Tomcat;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.servlet.ServletContextInitializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
```

```
public class MyCloudFoundryConfiguration {

    @Bean
    public TomcatServletWebServerFactory servletWebServerFactory() {
        return new TomcatServletWebServerFactory() {

            @Override
            protected void prepareContext(Host host, ServletContextInitializer[]
initializers) {
                super.prepareContext(host, initializers);
                StandardContext child = new StandardContext();
                child.addLifecycleListener(new Tomcat.FixContextListener());
                child.setPath("/cloudfoundryapplication");
                ServletContainerInitializer initializer =
getServletContextInitializer(getContextPath());
                child.addServletContainerInitializer(initializer,
Collections.emptySet());
                child.setCrossContext(true);
                host.addChild(child);
            }

        };
    }

    private ServletContainerInitializer getServletContextInitializer(String
contextPath) {
        return (classes, context) -> {
            Servlet servlet = new GenericServlet() {

                @Override
                public void service(ServletRequest req, ServletResponse res) throws
ServletException, IOException {
                    ServletContext context =
req.getServletContext().getContext(contextPath);

context.getRequestDispatcher("/cloudfoundryapplication").forward(req, res);
                }

            };
            context.addServlet("cloudfoundry", servlet).addMapping("/*");
        };
    }

}
```

Kotlin

```
import jakarta.servlet.GenericServlet
import jakarta.servlet.Servlet
import jakarta.servlet.ServletContainerInitializer
import jakarta.servlet.ServletContext
```

```
import jakarta.servlet.ServletException
import jakarta.servlet.ServletRequest
import jakarta.servlet.ServletResponse
import org.apache.catalina.Host
import org.apache.catalina.core.StandardContext
import org.apache.catalina.startup.Tomcat.FixContextListener
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory
import org.springframework.boot.web.servlet.ServletContextInitializer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import java.io.IOException
import java.util.Collections.emptySet

@Configuration(proxyBeanMethods = false)
class MyCloudFoundryConfiguration {

    @Bean
    fun servletWebServerFactory(): TomcatServletWebServerFactory {
        return object : TomcatServletWebServerFactory() {

            override fun prepareContext(host: Host, initializers:
Array<ServletContextInitializer>) {
                super.prepareContext(host, initializers)
                val child = StandardContext()
                child.addLifecycleListener(FixContextListener())
                child.path = "/cloudfoundryapplication"
                val initializer = getServletContextInitializer(contextPath)
                child.addServletContainerInitializer(initializer, emptySet())
                child.crossContext = true
                host.addChild(child)
            }

        }
    }

    private fun getServletContextInitializer(contextPath: String):
ServletContainerInitializer {
        return ServletContainerInitializer { classes: Set<Class<*>?>?, context:
ServletContext ->
            val servlet: Servlet = object : GenericServlet() {

                @Throws(ServletException::class, IOException::class)
                override fun service(req: ServletRequest, res: ServletResponse) {
                    val servletContext = req.servletContext.getContext(contextPath)

servletContext.getRequestDispatcher("/cloudfoundryapplication").forward(req, res)
                }

            }
            context.addServlet("cloudfoundry", servlet).addMapping("/*")
        }
```

```
    }
}
```

If you're using a Webflux based application, you can use the following configuration:

Java

```
import java.util.Map;

import reactor.core.publisher.Mono;

import org.springframework.boot.autoconfigure.web.reactive.WebFluxProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.server.reactive.ContextPathCompositeHandler;
import org.springframework.http.server.reactive.HttpHandler;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.http.server.reactive.ServerHttpResponse;
import org.springframework.web.server.adapter.WebHttpHandlerBuilder;

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(WebFluxProperties.class)
public class MyReactiveCloudFoundryConfiguration {

    @Bean
    public HttpHandler httpHandler(ApplicationContext applicationContext,
WebFluxProperties properties) {
        HttpHandler httpHandler =
WebHttpHandlerBuilder.applicationContext(applicationContext).build();
        return new CloudFoundryHttpHandler(properties.getBasePath(), httpHandler);
    }

    private static final class CloudFoundryHttpHandler implements HttpHandler {

        private final HttpHandler delegate;

        private final ContextPathCompositeHandler contextPathDelegate;

        private CloudFoundryHttpHandler(String basePath, HttpHandler delegate) {
            this.delegate = delegate;
            this.contextPathDelegate = new
ContextPathCompositeHandler(Map.of(basePath, delegate));
        }

        @Override
        public Mono<Void> handle(ServerHttpRequest request, ServerHttpResponse
response) {
            // Remove underlying context path first (e.g. Servlet container)
            String path = request.getPath().pathWithinApplication().value();
```

```
            if (path.startsWith("/cloudfoundryapplication")) {
                return this.delegate.handle(request, response);
            }
            else {
                return this.contextPathDelegate.handle(request, response);
            }
        }

    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.web.reactive.WebFluxProperties
import org.springframework.boot.context.properties.EnableConfigurationProperties
import org.springframework.context.ApplicationContext
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.server.reactive.ContextPathCompositeHandler
import org.springframework.http.server.reactive.HttpHandler
import org.springframework.http.server.reactive.ServerHttpRequest
import org.springframework.http.server.reactive.ServerHttpResponse
import org.springframework.web.server.adapter.WebHttpHandlerBuilder
import reactor.core.publisher.Mono

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(WebFluxProperties::class)
class MyReactiveCloudFoundryConfiguration {

    @Bean
    fun httpHandler(applicationContext: ApplicationContext, properties:
WebFluxProperties): HttpHandler {
        val httpHandler =
WebHttpHandlerBuilder.applicationContext(applicationContext).build()
        return CloudFoundryHttpHandler(properties.basePath, httpHandler)
    }

    private class CloudFoundryHttpHandler(basePath: String, private val delegate:
HttpHandler) : HttpHandler {
        private val contextPathDelegate = ContextPathCompositeHandler(mapOf(basePath
to delegate))

        override fun handle(request: ServerHttpRequest, response: ServerHttpResponse):
Mono<Void> {
            // Remove underlying context path first (e.g. Servlet container)
            val path = request.path.pathWithinApplication().value()
            return if (path.startsWith("/cloudfoundryapplication")) {
                delegate.handle(request, response)
            } else {
                contextPathDelegate.handle(request, response)
            }
        }
    }
}
```

13.13. What to Read Next

You might want to read about graphing tools such as Graphite.

Otherwise, you can continue on to read about "deployment options" or jump ahead for some in-depth information about Spring Boot's build tool plugins.

Chapter 14. Deploying Spring Boot Applications

Spring Boot's flexible packaging options provide a great deal of choice when it comes to deploying your application. You can deploy Spring Boot applications to a variety of cloud platforms, to virtual/real machines, or make them fully executable for Unix systems.

This section covers some of the more common deployment scenarios.

14.1. Deploying to the Cloud

Spring Boot's executable jars are ready-made for most popular cloud PaaS (Platform-as-a-Service) providers. These providers tend to require that you "bring your own container". They manage application processes (not Java applications specifically), so they need an intermediary layer that adapts *your* application to the *cloud's* notion of a running process.

Two popular cloud providers, Heroku and Cloud Foundry, employ a "buildpack" approach. The buildpack wraps your deployed code in whatever is needed to *start* your application. It might be a JDK and a call to `java`, an embedded web server, or a full-fledged application server. A buildpack is pluggable, but ideally you should be able to get by with as few customizations to it as possible. This reduces the footprint of functionality that is not under your control. It minimizes divergence between development and production environments.

Ideally, your application, like a Spring Boot executable jar, has everything that it needs to run packaged within it.

In this section, we look at what it takes to get the application that we developed in the "Getting Started" section up and running in the Cloud.

14.1.1. Cloud Foundry

Cloud Foundry provides default buildpacks that come into play if no other buildpack is specified. The Cloud Foundry Java buildpack has excellent support for Spring applications, including Spring Boot. You can deploy stand-alone executable jar applications as well as traditional `.war` packaged applications.

Once you have built your application (by using, for example, `mvn clean package`) and have installed the `cf` command line tool, deploy your application by using the `cf push` command, substituting the path to your compiled `.jar`. Be sure to have logged in with your `cf` command line client before pushing an application. The following line shows using the `cf push` command to deploy an application:

```
$ cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

NOTE In the preceding example, we substitute `acloudyspringtime` for whatever value you give `cf` as the name of your application.

See the `cf push` documentation for more options. If there is a Cloud Foundry `manifest.yml` file present in the same directory, it is considered.

At this point, `cf` starts uploading your application, producing output similar to the following example:

```
Uploading acloudyspringtime... OK
Preparing to start acloudyspringtime... OK
-----> Downloaded app package (8.9M)
-----> Java Buildpack Version: v3.12 (offline) | https://github.com/cloudfoundry/java-
buildpack.git#6f25b7e
-----> Downloading Open Jdk JRE
       Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.6s)
-----> Downloading Open JDK Like Memory Calculator 2.0.2_RELEASE from https://java-
buildpack.cloudfoundry.org/memory-calculator/trusty/x86_64/memory-calculator-
2.0.2_RELEASE.tar.gz (found in cache)
       Memory Settings: -Xss349K -Xmx681574K -XX:MaxMetaspaceSize=104857K -Xms681574K
-XX:MetaspaceSize=104857K
-----> Downloading Container Certificate Trust Store 1.0.0_RELEASE from https://java-
buildpack.cloudfoundry.org/container-certificate-trust-store/container-certificate-
trust-store-1.0.0_RELEASE.jar (found in cache)
       Adding certificates to .java-
buildpack/container_certificate_trust_store/truststore.jks (0.6s)
-----> Downloading Spring Auto Reconfiguration 1.10.0_RELEASE from https://java-
buildpack.cloudfoundry.org/auto-reconfiguration/auto-reconfiguration-
1.10.0_RELEASE.jar (found in cache)
Checking status of app 'acloudyspringtime'...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 starting)
  ...
  1 of 1 instances running (1 running)

App started
```

Congratulations! The application is now live!

Once your application is live, you can verify the status of the deployed application by using the `cf apps` command, as shown in the following example:

```
$ cf apps
Getting applications in ...
OK

name                requested state   instances   memory   disk   urls
...
acloudyspringtime   started           1/1         512M     1G
acloudyspringtime.cfapps.io
...
```

Once Cloud Foundry acknowledges that your application has been deployed, you should be able to find the application at the URI given. In the preceding example, you could find it at https://acloudyspringtime.cfapps.io/.

Binding to Services

By default, metadata about the running application as well as service connection information is exposed to the application as environment variables (for example: `$VCAP_SERVICES`). This architecture decision is due to Cloud Foundry's polyglot (any language and platform can be supported as a buildpack) nature. Process-scoped environment variables are language agnostic.

Environment variables do not always make for the easiest API, so Spring Boot automatically extracts them and flattens the data into properties that can be accessed through Spring's `Environment` abstraction, as shown in the following example:

Java

```
import org.springframework.context.EnvironmentAware;
import org.springframework.core.env.Environment;
import org.springframework.stereotype.Component;

@Component
public class MyBean implements EnvironmentAware {

    private String instanceId;

    @Override
    public void setEnvironment(Environment environment) {
        this.instanceId = environment.getProperty("vcap.application.instance_id");
    }

    // ...

}
```

Kotlin

```
import org.springframework.context.EnvironmentAware
import org.springframework.core.env.Environment
import org.springframework.stereotype.Component

@Component
class MyBean : EnvironmentAware {

    private var instanceId: String? = null

    override fun setEnvironment(environment: Environment) {
        instanceId = environment.getProperty("vcap.application.instance_id")
    }

    // ...

}
```

All Cloud Foundry properties are prefixed with `vcap`. You can use `vcap` properties to access application information (such as the public URL of the application) and service information (such as database credentials). See the `CloudFoundryVcapEnvironmentPostProcessor` Javadoc for complete details.

TIP The Java CFEnv project is a better fit for tasks such as configuring a DataSource.

14.1.2. Kubernetes

Spring Boot auto-detects Kubernetes deployment environments by checking the environment for `"*_SERVICE_HOST"` and `"*_SERVICE_PORT"` variables. You can override this detection with the `spring.main.cloud-platform` configuration property.

Spring Boot helps you to manage the state of your application and export it with HTTP Kubernetes Probes using Actuator.

Kubernetes Container Lifecycle

When Kubernetes deletes an application instance, the shutdown process involves several subsystems concurrently: shutdown hooks, unregistering the service, removing the instance from the load-balancer… Because this shutdown processing happens in parallel (and due to the nature of distributed systems), there is a window during which traffic can be routed to a pod that has also begun its shutdown processing.

You can configure a sleep execution in a preStop handler to avoid requests being routed to a pod that has already begun shutting down. This sleep should be long enough for new requests to stop being routed to the pod and its duration will vary from deployment to deployment. The preStop handler can be configured by using the PodSpec in the pod's configuration file as follows:

```
spec:
  containers:
  - name: "example-container"
    image: "example-image"
    lifecycle:
      preStop:
        exec:
          command: ["sh", "-c", "sleep 10"]
```

Once the pre-stop hook has completed, SIGTERM will be sent to the container and graceful shutdown will begin, allowing any remaining in-flight requests to complete.

NOTE When Kubernetes sends a SIGTERM signal to the pod, it waits for a specified time called the termination grace period (the default for which is 30 seconds). If the containers are still running after the grace period, they are sent the SIGKILL signal and forcibly removed. If the pod takes longer than 30 seconds to shut down, which could be because you have increased `spring.lifecycle.timeout-per-shutdown-phase`, make sure to increase the termination grace period by setting the `terminationGracePeriodSeconds` option in the Pod YAML.

14.1.3. Heroku

Heroku is another popular PaaS platform. To customize Heroku builds, you provide a `Procfile`, which provides the incantation required to deploy an application. Heroku assigns a `port` for the Java application to use and then ensures that routing to the external URI works.

You must configure your application to listen on the correct port. The following example shows the `Procfile` for our starter REST application:

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
```

Spring Boot makes `-D` arguments available as properties accessible from a Spring `Environment` instance. The `server.port` configuration property is fed to the embedded Tomcat, Jetty, or Undertow instance, which then uses the port when it starts up. The `$PORT` environment variable is assigned to us by the Heroku PaaS.

This should be everything you need. The most common deployment workflow for Heroku deployments is to `git push` the code to production, as shown in the following example:

```
$ git push heroku main
```

Which will result in the following:

```
Initializing repository, done.
Counting objects: 95, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (78/78), done.
Writing objects: 100% (95/95), 8.66 MiB | 606.00 KiB/s, done.
Total 95 (delta 31), reused 0 (delta 0)

-----> Java app detected
-----> Installing OpenJDK... done
-----> Installing Maven... done
-----> Installing settings.xml... done
-----> Executing: mvn -B -DskipTests=true clean install

       [INFO] Scanning for projects...
       Downloading: https://repo.spring.io/...
       Downloaded: https://repo.spring.io/... (818 B at 1.8 KB/sec)
        ....
       Downloaded: https://s3pository.heroku.com/jvm/... (152 KB at 595.3 KB/sec)
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/target/...
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/pom.xml ...
       [INFO] ------------------------------------------------------------------------
       [INFO] BUILD SUCCESS
       [INFO] ------------------------------------------------------------------------
       [INFO] Total time: 59.358s
       [INFO] Finished at: Fri Mar 07 07:28:25 UTC 2014
       [INFO] Final Memory: 20M/493M
       [INFO] ------------------------------------------------------------------------

-----> Discovering process types
       Procfile declares types -> web

-----> Compressing... done, 70.4MB
-----> Launching... done, v6
       https://agile-sierra-1405.herokuapp.com/ deployed to Heroku

To git@heroku.com:agile-sierra-1405.git
 * [new branch]      main -> main
```

Your application should now be up and running on Heroku. For more details, see Deploying Spring Boot Applications to Heroku.

14.1.4. OpenShift

OpenShift has many resources describing how to deploy Spring Boot applications, including:

- Using the S2I builder
- Architecture guide
- Running as a traditional web application on Wildfly

- OpenShift Commons Briefing

14.1.5. Amazon Web Services (AWS)

Amazon Web Services offers multiple ways to install Spring Boot-based applications, either as traditional web applications (war) or as executable jar files with an embedded web server. The options include:

- AWS Elastic Beanstalk
- AWS Code Deploy
- AWS OPS Works
- AWS Cloud Formation
- AWS Container Registry

Each has different features and pricing models. In this document, we describe to approach using AWS Elastic Beanstalk.

AWS Elastic Beanstalk

As described in the official Elastic Beanstalk Java guide, there are two main options to deploy a Java application. You can either use the "Tomcat Platform" or the "Java SE platform".

Using the Tomcat Platform

This option applies to Spring Boot projects that produce a war file. No special configuration is required. You need only follow the official guide.

Using the Java SE Platform

This option applies to Spring Boot projects that produce a jar file and run an embedded web container. Elastic Beanstalk environments run an nginx instance on port 80 to proxy the actual application, running on port 5000. To configure it, add the following line to your `application.properties` file:

```
server.port=5000
```

TIP

Upload binaries instead of sources

By default, Elastic Beanstalk uploads sources and compiles them in AWS. However, it is best to upload the binaries instead. To do so, add lines similar to the following to your `.elasticbeanstalk/config.yml` file:

```
deploy:
    artifact: target/demo-0.0.1-SNAPSHOT.jar
```

TIP

Reduce costs by setting the environment type

By default an Elastic Beanstalk environment is load balanced. The load balancer has a significant cost. To avoid that cost, set the environment type to "Single instance", as described in the Amazon documentation. You can also create single instance environments by using the CLI and the following command:

```
eb create -s
```

Summary

This is one of the easiest ways to get to AWS, but there are more things to cover, such as how to integrate Elastic Beanstalk into any CI / CD tool, use the Elastic Beanstalk Maven plugin instead of the CLI, and others. There is a blog post covering these topics more in detail.

14.1.6. CloudCaptain and Amazon Web Services

CloudCaptain works by turning your Spring Boot executable jar or war into a minimal VM image that can be deployed unchanged either on VirtualBox or on AWS. CloudCaptain comes with deep integration for Spring Boot and uses the information from your Spring Boot configuration file to automatically configure ports and health check URLs. CloudCaptain leverages this information both for the images it produces as well as for all the resources it provisions (instances, security groups, elastic load balancers, and so on).

Once you have created a CloudCaptain account, connected it to your AWS account, installed the latest version of the CloudCaptain Client, and ensured that the application has been built by Maven or Gradle (by using, for example, `mvn clean package`), you can deploy your Spring Boot application to AWS with a command similar to the following:

```
$ boxfuse run myapp-1.0.jar -env=prod
```

See the `boxfuse run` documentation for more options. If there is a `boxfuse.conf` file present in the current directory, it is considered.

TIP

By default, CloudCaptain activates a Spring profile named `boxfuse` on startup. If your executable jar or war contains an `application-boxfuse.properties` file, CloudCaptain bases its configuration on the properties it contains.

At this point, CloudCaptain creates an image for your application, uploads it, and configures and starts the necessary resources on AWS, resulting in output similar to the following example:

```
Fusing Image for myapp-1.0.jar ...
Image fused in 00:06.838s (53937 K) -> axelfontaine/myapp:1.0
Creating axelfontaine/myapp ...
Pushing axelfontaine/myapp:1.0 ...
Verifying axelfontaine/myapp:1.0 ...
Creating Elastic IP ...
Mapping myapp-axelfontaine.boxfuse.io to 52.28.233.167 ...
Waiting for AWS to create an AMI for axelfontaine/myapp:1.0 in eu-central-1 (this may
take up to 50 seconds) ...
AMI created in 00:23.557s -> ami-d23f38cf
Creating security group boxfuse-sg_axelfontaine/myapp:1.0 ...
Launching t2.micro instance of axelfontaine/myapp:1.0 (ami-d23f38cf) in eu-central-1
...
Instance launched in 00:30.306s -> i-92ef9f53
Waiting for AWS to boot Instance i-92ef9f53 and Payload to start at
https://52.28.235.61/ ...
Payload started in 00:29.266s -> https://52.28.235.61/
Remapping Elastic IP 52.28.233.167 to i-92ef9f53 ...
Waiting 15s for AWS to complete Elastic IP Zero Downtime transition ...
Deployment completed successfully. axelfontaine/myapp:1.0 is up and running at
https://myapp-axelfontaine.boxfuse.io/
```

Your application should now be up and running on AWS.

See the blog post on deploying Spring Boot apps on EC2 as well as the documentation for the CloudCaptain Spring Boot integration to get started with a Maven build to run the app.

14.1.7. Azure

This Getting Started guide walks you through deploying your Spring Boot application to either Azure Spring Cloud or Azure App Service.

14.1.8. Google Cloud

Google Cloud has several options that can be used to launch Spring Boot applications. The easiest to get started with is probably App Engine, but you could also find ways to run Spring Boot in a container with Container Engine or on a virtual machine with Compute Engine.

To deploy your first app to App Engine standard environment, follow this tutorial.

Alternatively, App Engine Flex requires you to create an `app.yaml` file to describe the resources your app requires. Normally, you put this file in `src/main/appengine`, and it should resemble the following file:

```
service: "default"

runtime: "java17"
env: "flex"

handlers:
- url: "/.*"
  script: "this field is required, but ignored"

manual_scaling:
  instances: 1

health_check:
  enable_health_check: false

env_variables:
  ENCRYPT_KEY: "your_encryption_key_here"
```

You can deploy the app (for example, with a Maven plugin) by adding the project ID to the build configuration, as shown in the following example:

```
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>appengine-maven-plugin</artifactId>
    <version>2.4.4</version>
    <configuration>
        <project>myproject</project>
    </configuration>
</plugin>
```

Then deploy with `mvn appengine:deploy` (you need to authenticate first, otherwise the build fails).

14.2. Installing Spring Boot Applications

In addition to running Spring Boot applications by using `java -jar` directly, it is also possible to run them as `systemd`, `init.d` or Windows services.

14.2.1. Installation as a systemd Service

`systemd` is the successor of the System V init system and is now being used by many modern Linux distributions. Spring Boot applications can be launched by using `systemd` 'service' scripts.

Assuming that you have a Spring Boot application packaged as an uber jar in `/var/myapp`, to install it as a `systemd` service, create a script named `myapp.service` and place it in `/etc/systemd/system` directory. The following script offers an example:

```
[Unit]
Description=myapp
After=syslog.target network.target

[Service]
User=myapp
Group=myapp

Environment="JAVA_HOME=/path/to/java/home"

ExecStart=${JAVA_HOME}/bin/java -jar /var/myapp/myapp.jar
ExecStop=/bin/kill -15 $MAINPID
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```

IMPORTANT	Remember to change the `Description`, `User`, `Group`, `Environment` and `ExecStart` fields for your application.

NOTE	The `ExecStart` field does not declare the script action command, which means that the `run` command is used by default.

The user that runs the application, the PID file, and the console log file are managed by `systemd` itself and therefore must be configured by using appropriate fields in the 'service' script. Consult the service unit configuration man page for more details.

To flag the application to start automatically on system boot, use the following command:

```
$ systemctl enable myapp.service
```

Run `man systemctl` for more details.

14.2.2. Installation as an init.d Service (System V)

To use your application as `init.d` service, configure its build to produce a fully executable jar.

CAUTION	Fully executable jars work by embedding an extra script at the front of the file. Currently, some tools do not accept this format, so you may not always be able to use this technique. For example, `jar -xf` may silently fail to extract a jar or war that has been made fully executable. It is recommended that you make your jar or war fully executable only if you intend to execute it directly, rather than running it with `java -jar` or deploying it to a servlet container.

CAUTION	A zip64-format jar file cannot be made fully executable. Attempting to do so will result in a jar file that is reported as corrupt when executed directly or with `java -jar`. A standard-format jar file that contains one or more zip64-format nested jars can be fully executable.

To create a 'fully executable' jar with Maven, use the following plugin configuration:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <executable>true</executable>
    </configuration>
</plugin>
```

The following example shows the equivalent Gradle configuration:

```
tasks.named('bootJar') {
    launchScript()
}
```

It can then be symlinked to `init.d` to support the standard `start`, `stop`, `restart`, and `status` commands.

The default launch script that is added to a fully executable jar supports most Linux distributions and is tested on CentOS and Ubuntu. Other platforms, such as OS X and FreeBSD, require the use of a custom script. The default scripts supports the following features:

- Starts the services as the user that owns the jar file
- Tracks the application's PID by using `/var/run/<appname>/<appname>.pid`
- Writes console logs to `/var/log/<appname>.log`

Assuming that you have a Spring Boot application installed in `/var/myapp`, to install a Spring Boot application as an `init.d` service, create a symlink, as follows:

```
$ sudo ln -s /var/myapp/myapp.jar /etc/init.d/myapp
```

Once installed, you can start and stop the service in the usual way. For example, on a Debian-based system, you could start it with the following command:

```
$ service myapp start
```

TIP	If your application fails to start, check the log file written to `/var/log/<appname>.log` for errors.

You can also flag the application to start automatically by using your standard operating system tools. For example, on Debian, you could use the following command:

```
$ update-rc.d myapp defaults <priority>
```

Securing an init.d Service

NOTE	The following is a set of guidelines on how to secure a Spring Boot application that runs as an init.d service. It is not intended to be an exhaustive list of everything that should be done to harden an application and the environment in which it runs.

When executed as root, as is the case when root is being used to start an init.d service, the default executable script runs the application as the user specified in the `RUN_AS_USER` environment variable. When the environment variable is not set, the user who owns the jar file is used instead. You should never run a Spring Boot application as `root`, so `RUN_AS_USER` should never be root and your application's jar file should never be owned by root. Instead, create a specific user to run your application and set the `RUN_AS_USER` environment variable or use `chown` to make it the owner of the jar file, as shown in the following example:

```
$ chown bootapp:bootapp your-app.jar
```

In this case, the default executable script runs the application as the `bootapp` user.

TIP	To reduce the chances of the application's user account being compromised, you should consider preventing it from using a login shell. For example, you can set the account's shell to `/usr/sbin/nologin`.

You should also take steps to prevent the modification of your application's jar file. Firstly, configure its permissions so that it cannot be written and can only be read or executed by its owner, as shown in the following example:

```
$ chmod 500 your-app.jar
```

Second, you should also take steps to limit the damage if your application or the account that is running it is compromised. If an attacker does gain access, they could make the jar file writable and change its contents. One way to protect against this is to make it immutable by using `chattr`, as shown in the following example:

```
$ sudo chattr +i your-app.jar
```

This will prevent any user, including root, from modifying the jar.

If root is used to control the application's service and you use a `.conf` file to customize its startup, the `.conf` file is read and evaluated by the root user. It should be secured accordingly. Use `chmod` so

that the file can only be read by the owner and use `chown` to make root the owner, as shown in the following example:

```
$ chmod 400 your-app.conf
$ sudo chown root:root your-app.conf
```

Customizing the Startup Script

The default embedded startup script written by the Maven or Gradle plugin can be customized in a number of ways. For most people, using the default script along with a few customizations is usually enough. If you find you cannot customize something that you need to, use the `embeddedLaunchScript` option to write your own file entirely.

Customizing the Start Script When It Is Written

It often makes sense to customize elements of the start script as it is written into the jar file. For example, init.d scripts can provide a "description". Since you know the description up front (and it need not change), you may as well provide it when the jar is generated.

To customize written elements, use the `embeddedLaunchScriptProperties` option of the Spring Boot Maven plugin or the `properties` property of the Spring Boot Gradle plugin's `launchScript`.

The following property substitutions are supported with the default script:

Name	Description	Gradle default	Maven default
`mode`	The script mode.	`auto`	`auto`
`initInfoProvides`	The `Provides` section of "INIT INFO"	`${task.baseName}`	`${project.artifactId}`
`initInfoRequiredStart`	`Required-Start` section of "INIT INFO".	`$remote_fs $syslog $network`	`$remote_fs $syslog $network`
`initInfoRequiredStop`	`Required-Stop` section of "INIT INFO".	`$remote_fs $syslog $network`	`$remote_fs $syslog $network`
`initInfoDefaultStart`	`Default-Start` section of "INIT INFO".	`2 3 4 5`	`2 3 4 5`
`initInfoDefaultStop`	`Default-Stop` section of "INIT INFO".	`0 1 6`	`0 1 6`
`initInfoShortDescription`	`Short-Description` section of "INIT INFO".	Single-line version of `${project.description}` (falling back to `${task.baseName}`)	`${project.name}`
`initInfoDescription`	`Description` section of "INIT INFO".	`${project.description}` (falling back to `${task.baseName}`)	`${project.description}` (falling back to `${project.name}`)

Name	Description	Gradle default	Maven default
`initInfoChkconfig`	`chkconfig` section of "INIT INFO"	`2345 99 01`	`2345 99 01`
`confFolder`	The default value for `CONF_FOLDER`	Folder containing the jar	Folder containing the jar
`inlinedConfScript`	Reference to a file script that should be inlined in the default launch script. This can be used to set environmental variables such as `JAVA_OPTS` before any external config files are loaded		
`logFolder`	Default value for `LOG_FOLDER`. Only valid for an `init.d` service		
`logFilename`	Default value for `LOG_FILENAME`. Only valid for an `init.d` service		
`pidFolder`	Default value for `PID_FOLDER`. Only valid for an `init.d` service		
`pidFilename`	Default value for the name of the PID file in `PID_FOLDER`. Only valid for an `init.d` service		
`useStartStopDaemon`	Whether the `start-stop-daemon` command, when it is available, should be used to control the process	`true`	`true`
`stopWaitTime`	Default value for `STOP_WAIT_TIME` in seconds. Only valid for an `init.d` service	60	60

Customizing a Script When It Runs

For items of the script that need to be customized *after* the jar has been written, you can use environment variables or a config file.

The following environment properties are supported with the default script:

Variable	Description
MODE	The “mode” of operation. The default depends on the way the jar was built but is usually auto (meaning it tries to guess if it is an init script by checking if it is a symlink in a directory called init.d). You can explicitly set it to service so that the stop|start|status|restart commands work or to run if you want to run the script in the foreground.
RUN_AS_USER	The user that will be used to run the application. When not set, the user that owns the jar file will be used.
USE_START_STOP_DAEMON	Whether the start-stop-daemon command, when it is available, should be used to control the process. Defaults to true.
PID_FOLDER	The root name of the pid folder (/var/run by default).
LOG_FOLDER	The name of the folder in which to put log files (/var/log by default).
CONF_FOLDER	The name of the folder from which to read .conf files (same folder as jar-file by default).
LOG_FILENAME	The name of the log file in the LOG_FOLDER (<appname>.log by default).
APP_NAME	The name of the app. If the jar is run from a symlink, the script guesses the app name. If it is not a symlink or you want to explicitly set the app name, this can be useful.
RUN_ARGS	The arguments to pass to the program (the Spring Boot app).
JAVA_HOME	The location of the java executable is discovered by using the PATH by default, but you can set it explicitly if there is an executable file at $JAVA_HOME/bin/java.
JAVA_OPTS	Options that are passed to the JVM when it is launched.
JARFILE	The explicit location of the jar file, in case the script is being used to launch a jar that it is not actually embedded.
DEBUG	If not empty, sets the -x flag on the shell process, allowing you to see the logic in the script.
STOP_WAIT_TIME	The time in seconds to wait when stopping the application before forcing a shutdown (60 by default).

NOTE The PID_FOLDER, LOG_FOLDER, and LOG_FILENAME variables are only valid for an init.d service. For systemd, the equivalent customizations are made by using the ‘service’ script. See the service unit configuration man page for more details.

Using a Conf Gile

With the exception of JARFILE and APP_NAME, the settings listed in the preceding section can be configured by using a .conf file. The file is expected to be next to the jar file and have the same name but suffixed with .conf rather than .jar. For example, a jar named /var/myapp/myapp.jar uses the configuration file named /var/myapp/myapp.conf, as shown in the following example:

myapp.conf

```
JAVA_OPTS=-Xmx1024M
LOG_FOLDER=/custom/log/folder
```

TIP	If you do not like having the config file next to the jar file, you can set a `CONF_FOLDER` environment variable to customize the location of the config file.

To learn about securing this file appropriately, see the guidelines for securing an init.d service.

14.2.3. Microsoft Windows Services

A Spring Boot application can be started as a Windows service by using `winsw`.

A (separately maintained sample) describes step-by-step how you can create a Windows service for your Spring Boot application.

14.3. Efficient deployments

14.3.1. Unpacking the Executable JAR

If you are running your application from a container, you can use an executable jar, but it is also often an advantage to explode it and run it in a different way. Certain PaaS implementations may also choose to unpack archives before they run. For example, Cloud Foundry operates this way. One way to run an unpacked archive is by starting the appropriate launcher, as follows:

```
$ jar -xf myapp.jar
$ java org.springframework.boot.loader.JarLauncher
```

This is actually slightly faster on startup (depending on the size of the jar) than running from an unexploded archive. After startup, you should not expect any differences.

Once you have unpacked the jar file, you can also get an extra boost to startup time by running the app with its "natural" main method instead of the `JarLauncher`. For example:

```
$ jar -xf myapp.jar
$ java -cp "BOOT-INF/classes:BOOT-INF/lib/*" com.example.MyApplication
```

NOTE	Using the `JarLauncher` over the application's main method has the added benefit of a predictable classpath order. The jar contains a `classpath.idx` file which is used by the `JarLauncher` when constructing the classpath.

14.3.2. Using Ahead-of-time Processing With the JVM

It's beneficial for the startup time to run your application using the AOT generated initialization

code. First, you need to ensure that the jar you are building includes AOT generated code.

For Maven, this means that you should build with `-Pnative` to activate the `native` profile:

```
$ mvn -Pnative package
```

For Gradle, you need to ensure that your build includes the `org.springframework.boot.aot` plugin.

When the JAR has been built, run it with `spring.aot.enabled` system property set to `true`. For example:

```
$ java -Dspring.aot.enabled=true -jar myapplication.jar

........ Starting AOT-processed MyApplication ...
```

Beware that using the ahead-of-time processing has drawbacks. It implies the following restrictions:

- The classpath is fixed and fully defined at build time
- The beans defined in your application cannot change at runtime, meaning:
  - The Spring `@Profile` annotation and profile-specific configuration have limitations.
  - Properties that change if a bean is created are not supported (for example, `@ConditionalOnProperty` and `.enable` properties).

To learn more about ahead-of-time processing, please see the Understanding Spring Ahead-of-Time Processing section.

14.4. What to Read Next

See the Cloud Foundry, Heroku, OpenShift, and Boxfuse web sites for more information about the kinds of features that a PaaS can offer. These are just four of the most popular Java PaaS providers. Since Spring Boot is so amenable to cloud-based deployment, you can freely consider other providers as well.

The next section goes on to cover the *GraalVM Native Images*, or you can jump ahead to read about the *Spring Boot CLI* or our *build tool plugins*.

Chapter 15. GraalVM Native Image Support

GraalVM Native Images are standalone executables that can be generated by processing compiled Java applications ahead-of-time. Native Images generally have a smaller memory footprint and start faster than their JVM counterparts.

15.1. Introducing GraalVM Native Images

GraalVM Native Images provide a new way to deploy and run Java applications. Compared to the Java Virtual Machine, native images can run with a smaller memory footprint and with much faster startup times.

They are well suited to applications that are deployed using container images and are especially interesting when combined with "Function as a service" (FaaS) platforms.

Unlike traditional applications written for the JVM, GraalVM Native Image applications require ahead-of-time processing in order to create an executable. This ahead-of-time processing involves statically analyzing your application code from its main entry point.

A GraalVM Native Image is a complete, platform-specific executable. You do not need to ship a Java Virtual Machine in order to run a native image.

TIP	If you just want to get started and experiment with GraalVM you can skip ahead to the "Developing Your First GraalVM Native Application" section and return to this section later.

15.1.1. Key Differences with JVM Deployments

The fact that GraalVM Native Images are produced ahead-of-time means that there are some key differences between native and JVM based applications. The main differences are:

- Static analysis of your application is performed at build-time from the `main` entry point.
- Code that cannot be reached when the native image is created will be removed and won't be part of the executable.
- GraalVM is not directly aware of dynamic elements of your code and must be told about reflection, resources, serialization, and dynamic proxies.
- The application classpath is fixed at build time and cannot change.
- There is no lazy class loading, everything shipped in the executables will be loaded in memory on startup.
- There are some limitations around some aspects of Java applications that are not fully supported.

On top of those differences, Spring uses a process called Spring Ahead-of-Time processing, which imposes further limitations. Please make sure to read at least the beginning of the next section to learn about those.

TIP	The Native Image Compatibility Guide section of the GraalVM reference documentation provides more details about GraalVM limitations.

15.1.2. Understanding Spring Ahead-of-Time Processing

Typical Spring Boot applications are quite dynamic and configuration is performed at runtime. In fact, the concept of Spring Boot auto-configuration depends heavily on reacting to the state of the runtime in order to configure things correctly.

Although it would be possible to tell GraalVM about these dynamic aspects of the application, doing so would undo most of the benefit of static analysis. So instead, when using Spring Boot to create native images, a closed-world is assumed and the dynamic aspects of the application are restricted.

A closed-world assumption implies, besides the limitations created by GraalVM itself, the following restrictions:

- The beans defined in your application cannot change at runtime, meaning:
  - The Spring `@Profile` annotation and profile-specific configuration have limitations.
  - Properties that change if a bean is created are not supported (for example, `@ConditionalOnProperty` and `.enable` properties).

When these restrictions are in place, it becomes possible for Spring to perform ahead-of-time processing during build-time and generate additional assets that GraalVM can use. A Spring AOT processed application will typically generate:

- Java source code
- Bytecode (for dynamic proxies etc)
- GraalVM JSON hint files:
  - Resource hints (`resource-config.json`)
  - Reflection hints (`reflect-config.json`)
  - Serialization hints (`serialization-config.json`)
  - Java Proxy Hints (`proxy-config.json`)
  - JNI Hints (`jni-config.json`)

Source Code Generation

Spring applications are composed of Spring Beans. Internally, Spring Framework uses two distinct concepts to manage beans. There are bean instances, which are the actual instances that have been created and can be injected into other beans. There are also bean definitions which are used to define attributes of a bean and how its instance should be created.

If we take a typical `@Configuration` class:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyConfiguration {

    @Bean
    public MyBean myBean() {
        return new MyBean();
    }

}
```

The bean definition is created by parsing the `@Configuration` class and finding the `@Bean` methods. In the above example, we're defining a `BeanDefinition` for a singleton bean named `myBean`. We're also creating a `BeanDefinition` for the `MyConfiguration` class itself.

When the `myBean` instance is required, Spring knows that it must invoke the `myBean()` method and use the result. When running on the JVM, `@Configuration` class parsing happens when your application starts and `@Bean` methods are invoked using reflection.

When creating a native image, Spring operates in a different way. Rather than parsing `@Configuration` classes and generating bean definitions at runtime, it does it at build-time. Once the bean definitions have been discovered, they are processed and converted into source code that can be analyzed by the GraalVM compiler.

The Spring AOT process would convert the configuration class above to code like this:

```
import org.springframework.beans.factory.aot.BeanInstanceSupplier;
import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.RootBeanDefinition;

/**
 * Bean definitions for {@link MyConfiguration}.
 */
public class MyConfiguration__BeanDefinitions {

    /**
     * Get the bean definition for 'myConfiguration'.
     */
    public static BeanDefinition getMyConfigurationBeanDefinition() {
        Class<?> beanType = MyConfiguration.class;
        RootBeanDefinition beanDefinition = new RootBeanDefinition(beanType);
        beanDefinition.setInstanceSupplier(MyConfiguration::new);
        return beanDefinition;
    }

    /**
     * Get the bean instance supplier for 'myBean'.
     */
    private static BeanInstanceSupplier<MyBean> getMyBeanInstanceSupplier() {
        return BeanInstanceSupplier.<MyBean>forFactoryMethod(MyConfiguration.class,
"myBean")
            .withGenerator((registeredBean) ->
registeredBean.getBeanFactory().getBean(MyConfiguration.class).myBean());
    }

    /**
     * Get the bean definition for 'myBean'.
     */
    public static BeanDefinition getMyBeanBeanDefinition() {
        Class<?> beanType = MyBean.class;
        RootBeanDefinition beanDefinition = new RootBeanDefinition(beanType);
        beanDefinition.setInstanceSupplier(getMyBeanInstanceSupplier());
        return beanDefinition;
    }

}
```

NOTE The exact code generated may differ depending on the nature of your bean definitions.

You can see above that the generated code creates equivalent bean definitions to the `@Configuration` class, but in a direct way that can be understood by GraalVM.

There is a bean definition for the `myConfiguration` bean, and one for `myBean`. When a `myBean` instance is required, a `BeanInstanceSupplier` is called. This supplier will invoke the `myBean()` method on the

`myConfiguration` bean.

NOTE	During Spring AOT processing your application is started up to the point that bean definitions are available. Bean instances are not created during the AOT processing phase.

Spring AOT will generate code like this for all your bean definitions. It will also generate code when bean post-processing is required (for example, to call `@Autowired` methods). An `ApplicationContextInitializer` will also be generated which will be used by Spring Boot to initialize the `ApplicationContext` when an AOT processed application is actually run.

TIP	Although AOT generated source code can be verbose, it is quite readable and can be helpful when debugging an application. Generated source files can be found in `target/spring-aot/main/sources` when using Maven and `build/generated/aotSources` with Gradle.

Hint File Generation

In addition to generating source files, the Spring AOT engine will also generate hint files that are used by GraalVM. Hint files contain JSON data that describes how GraalVM should deal with things that it can't understand by directly inspecting the code.

For example, you might be using a Spring annotation on a private method. Spring will need to use reflection in order to invoke private methods, even on GraalVM. When such situations arise, Spring can write a reflection hint so that GraalVM knows that even though the private method isn't called directly, it still needs to be available in the native image.

Hint files are generated under `META-INF/native-image` where they are automatically picked up by GraalVM.

TIP	Generated hint files can be found in `target/spring-aot/main/resources` when using Maven and `build/generated/aotResources` with Gradle.

Proxy Class Generation

Spring sometimes needs to generate proxy classes to enhance the code you've written with additional features. To do this, it uses the cglib library which directly generates bytecode.

When an application is running on the JVM, proxy classes are generated dynamically as the application runs. When creating a native image, these proxies need to be created at build-time so that they can be included by GraalVM.

NOTE	Unlike source code generation, generated bytecode isn't particularly helpful when debugging an application. However, if you need to inspect the contents of the `.class` files using a tool such as `javap` you can find them in `target/spring-aot/main/classes` for Maven and `build/generated/aotClasses` for Gradle.

15.2. Developing Your First GraalVM Native Application

Now that we have a good overview of GraalVM Native Images and how the Spring ahead-of-time engine works, we can look at how to create an application.

There are two main ways to build a Spring Boot native image application:

- Using Spring Boot support for Cloud Native Buildpacks to generate a lightweight container containing a native executable.
- Using GraalVM Native Build Tools to generate a native executable.

TIP	The easiest way to start a new native Spring Boot project is to go to start.spring.io, add the "GraalVM Native Support" dependency and generate the project. The included `HELP.md` file will provide getting started hints.

15.2.1. Sample Application

We need an example application that we can use to create our native image. For our purposes, the simple "Hello World!" web application that's covered in the "Developing Your First Spring Boot Application" section will suffice.

To recap, our main application code looks like this:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class MyApplication {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

This application uses Spring MVC and embedded Tomcat, both of which have been tested and verified to work with GraalVM native images.

15.2.2. Building a Native Image Using Buildpacks

Spring Boot includes buildpack support for native images directly for both Maven and Gradle. This means you can just type a single command and quickly get a sensible image into your locally running Docker daemon. The resulting image doesn't contain a JVM, instead the native image is compiled statically. This leads to smaller images.

NOTE	The builder used for the images is `paketobuildpacks/builder-jammy-tiny:latest`. It has small footprint and reduced attack surface, but you can also use `paketobuildpacks/builder-jammy-base:latest` or `paketobuildpacks/builder-jammy-full:latest` to have more tools available in the image if required.

System Requirements

Docker should be installed. See Get Docker for more details. Configure it to allow non-root user if you are on Linux.

NOTE	You can run `docker run hello-world` (without `sudo`) to check the Docker daemon is reachable as expected. Check the Maven or Gradle Spring Boot plugin documentation for more details.

TIP	On macOS, it is recommended to increase the memory allocated to Docker to at least `8GB`, and potentially add more CPUs as well. See this Stack Overflow answer for more details. On Microsoft Windows, make sure to enable the Docker WSL 2 backend for better performance.

Using Maven

To build a native image container using Maven you should ensure that your `pom.xml` file uses the `spring-boot-starter-parent` and the `org.graalvm.buildtools:native-maven-plugin`. You should have a `<parent>` section that looks like this:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.6</version>
</parent>
```

You additionally should have this in the `<build> <plugins>` section:

```
<plugin>
    <groupId>org.graalvm.buildtools</groupId>
    <artifactId>native-maven-plugin</artifactId>
</plugin>
```

The `spring-boot-starter-parent` declares a `native` profile that configures the executions that need to

run in order to create a native image. You can activate profiles using the `-P` flag on the command line.

TIP	If you don't want to use `spring-boot-starter-parent` you'll need to configure executions for the `process-aot` goal from Spring Boot's plugin and the `add-reachability-metadata` goal from the Native Build Tools plugin.

To build the image, you can run the `spring-boot:build-image` goal with the `native` profile active:

```
$ mvn -Pnative spring-boot:build-image
```

Using Gradle

The Spring Boot Gradle plugin automatically configures AOT tasks when the GraalVM Native Image plugin is applied. You should check that your Gradle build contains a `plugins` block that includes `org.graalvm.buildtools.native`.

As long as the `org.graalvm.buildtools.native` plugin is applied, the `bootBuildImage` task will generate a native image rather than a JVM one. You can run the task using:

```
$ gradle bootBuildImage
```

Running the example

Once you have run the appropriate build command, a Docker image should be available. You can start your application using `docker run`:

```
$ docker run --rm -p 8080:8080 docker.io/library/myproject:0.0.1-SNAPSHOT
```

You should see output similar to the following:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v3.1.6)
....... . . .
....... . . . (log output here)
....... . . .
........ Started MyApplication in 0.08 seconds (process running for 0.095)
```

NOTE	The startup time differs from machine to machine, but it should be much faster than a Spring Boot application running on a JVM.

If you open a web browser to `localhost:8080`, you should see the following output:

```
Hello World!
```

To gracefully exit the application, press `ctrl-c`.

15.2.3. Building a Native Image using Native Build Tools

If you want to generate a native executable directly without using Docker, you can use GraalVM Native Build Tools. Native Build Tools are plugins shipped by GraalVM for both Maven and Gradle. You can use them to perform a variety of GraalVM tasks, including generating a native image.

Prerequisites

To build a native image using the Native Build Tools, you'll need a GraalVM distribution on your machine. You can either download it manually on the Liberica Native Image Kit page, or you can use a download manager like SDKMAN!.

Linux and macOS

To install the native image compiler on macOS or Linux, we recommend using SDKMAN!. Get SDKMAN! from sdkman.io and install the Liberica GraalVM distribution by using the following commands:

```
$ sdk install java 22.3.r17-nik
$ sdk use java 22.3.r17-nik
```

Verify that the correct version has been configured by checking the output of `java -version`:

```
$ java -version
openjdk version "17.0.5" 2022-10-18 LTS
OpenJDK Runtime Environment GraalVM 22.3.0 (build 17.0.5+8-LTS)
OpenJDK 64-Bit Server VM GraalVM 22.3.0 (build 17.0.5+8-LTS, mixed mode)
```

Windows

On Windows, follow these instructions to install either GraalVM or Liberica Native Image Kit in version 22.3, the Visual Studio Build Tools and the Windows SDK. Due to the Windows related command-line maximum length, make sure to use x64 Native Tools Command Prompt instead of the regular Windows command line to run Maven or Gradle plugins.

Using Maven

As with the buildpack support, you need to make sure that you're using `spring-boot-starter-parent` in order to inherit the `native` profile and that the `org.graalvm.buildtools:native-maven-plugin` plugin is used.

With the `native` profile active, you can invoke the `native:compile` goal to trigger `native-image` compilation:

```
$ mvn -Pnative native:compile
```

The native image executable can be found in the `target` directory.

Using Gradle

When the Native Build Tools Gradle plugin is applied to your project, the Spring Boot Gradle plugin will automatically trigger the Spring AOT engine. Task dependencies are automatically configured, so you can just run the standard `nativeCompile` task to generate a native image:

```
$ gradle nativeCompile
```

The native image executable can be found in the `build/native/nativeCompile` directory.

Running the Example

At this point, your application should work. You can now start the application by running it directly:

Maven

```
$ target/myproject
```

Gradle

```
$ build/native/nativeCompile/myproject
```

You should see output similar to the following:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v3.1.6)
....... . . .
....... . . . (log output here)
....... . . .
........ Started MyApplication in 0.08 seconds (process running for 0.095)
```

NOTE	The startup time differs from machine to machine, but it should be much faster than a Spring Boot application running on a JVM.

If you open a web browser to `localhost:8080`, you should see the following output:

```
Hello World!
```

To gracefully exit the application, press `ctrl-c`.

15.3. Testing GraalVM Native Images

When writing native image applications, we recommend that you continue to use the JVM whenever possible to develop the majority of your unit and integration tests. This will help keep developer build times down and allow you to use existing IDE integrations. With broad test coverage on the JVM, you can then focus native image testing on the areas that are likely to be different.

For native image testing, you're generally looking to ensure that the following aspects work:

- The Spring AOT engine is able to process your application, and it will run in an AOT-processed mode.
- GraalVM has enough hints to ensure that a valid native image can be produced.

15.3.1. Testing Ahead-of-time Processing With the JVM

When a Spring Boot application runs, it attempts to detect if it is running as a native image. If it is running as a native image, it will initialize the application using the code that was generated during at build-time by the Spring AOT engine.

If the application is running on a regular JVM, then any AOT generated code is ignored.

Since the `native-image` compilation phase can take a while to complete, it's sometimes useful to run your application on the JVM but have it use the AOT generated initialization code. Doing so helps you to quickly validate that there are no errors in the AOT generated code and nothing is missing when your application is eventually converted to a native image.

To run a Spring Boot application on the JVM and have it use AOT generated code you can set the `spring.aot.enabled` system property to `true`.

For example:

```
$ java -Dspring.aot.enabled=true -jar myapplication.jar
```

NOTE You need to ensure that the jar you are testing includes AOT generated code. For Maven, this means that you should build with `-Pnative` to activate the `native` profile. For Gradle, you need to ensure that your build includes the `org.graalvm.buildtools.native` plugin.

If your application starts with the `spring.aot.enabled` property set to `true`, then you have higher

confidence that it will work when converted to a native image.

You can also consider running integration tests against the running application. For example, you could use the Spring `WebClient` to call your application REST endpoints. Or you might consider using a project like Selenium to check your application's HTML responses.

15.3.2. Testing With Native Build Tools

GraalVM Native Build Tools includes the ability to run tests inside a native image. This can be helpful when you want to deeply test that the internals of your application work in a GraalVM native image.

Generating the native image that contains the tests to run can be a time-consuming operation, so most developers will probably prefer to use the JVM locally. They can, however, be very useful as part of a CI pipeline. For example, you might choose to run native tests once a day.

Spring Framework includes ahead-of-time support for running tests. All the usual Spring testing features work with native image tests. For example, you can continue to use the `@SpringBootTest` annotation. You can also use Spring Boot test slices to test only specific parts of your application.

Spring Framework's native testing support works in the following way:

- Tests are analyzed in order to discover any `ApplicationContext` instances that will be required.
- Ahead-of-time processing is applied to each of these application contexts and assets are generated.
- A native image is created, with the generated assets being processed by GraalVM.
- The native image also includes the JUnit `TestEngine` configured with a list of the discovered tests.
- The native image is started, triggering the engine which will run each test and report results.

Using Maven

To run native tests using Maven, ensure that your `pom.xml` file uses the `spring-boot-starter-parent`. You should have a `<parent>` section that looks like this:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.6</version>
</parent>
```

The `spring-boot-starter-parent` declares a `nativeTest` profile that configures the executions that are needed to run the native tests. You can activate profiles using the `-P` flag on the command line.

TIP If you don't want to use `spring-boot-starter-parent` you'll need to configure executions for the `process-test-aot` goal from the Spring Boot plugin and the `test` goal from the Native Build Tools plugin.

To build the image and run the tests, use the `test` goal with the `nativeTest` profile active:

```
$ mvn -PnativeTest test
```

Using Gradle

The Spring Boot Gradle plugin automatically configures AOT test tasks when the GraalVM Native Image plugin is applied. You should check that your Gradle build contains a `plugins` block that includes `org.graalvm.buildtools.native`.

To run native tests using Gradle you can use the `nativeTest` task:

```
$ gradle nativeTest
```

15.4. Advanced Native Images Topics

15.4.1. Nested Configuration Properties

Reflection hints are automatically created for configuration properties by the Spring ahead-of-time engine. Nested configuration properties which are not inner classes, however, **must** be annotated with `@NestedConfigurationProperty`, otherwise they won't be detected and will not be bindable.

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.NestedConfigurationProperty;

@ConfigurationProperties(prefix = "my.properties")
public class MyProperties {

    private String name;

    @NestedConfigurationProperty
    private final Nested nested = new Nested();

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Nested getNested() {
        return this.nested;
    }

}
```

where `Nested` is:

```
public class Nested {

    private int number;

    public int getNumber() {
        return this.number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

}
```

The example above produces configuration properties for `my.properties.name` and `my.properties.nested.number`. Without the `@NestedConfigurationProperty` annotation on the `nested` field, the `my.properties.nested.number` property would not be bindable in a native image.

When using constructor binding, you have to annotate the field with `@NestedConfigurationProperty`:

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.NestedConfigurationProperty;

@ConfigurationProperties(prefix = "my.properties")
public class MyPropertiesCtor {

    private final String name;

    @NestedConfigurationProperty
    private final Nested nested;

    public MyPropertiesCtor(String name, Nested nested) {
        this.name = name;
        this.nested = nested;
    }

    public String getName() {
        return this.name;
    }

    public Nested getNested() {
        return this.nested;
    }

}
```

When using records, you have to annotate the parameter with `@NestedConfigurationProperty`:

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.NestedConfigurationProperty;

@ConfigurationProperties(prefix = "my.properties")
public record MyPropertiesRecord(String name, @NestedConfigurationProperty Nested
nested) {

}
```

When using Kotlin, you need to annotate the parameter of a data class with `@NestedConfigurationProperty`:

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.context.properties.NestedConfigurationProperty

@ConfigurationProperties(prefix = "my.properties")
data class MyPropertiesKotlin(
    val name: String,
    @NestedConfigurationProperty val nested: Nested
)
```

NOTE	Please use public getters and setters in all cases, otherwise the properties will not be bindable.

15.4.2. Converting a Spring Boot Executable Jar

It is possible to convert a Spring Boot executable jar into a native image as long as the jar contains the AOT generated assets. This can be useful for a number of reasons, including:

- You can keep your regular JVM pipeline and turn the JVM application into a native image on your CI/CD platform.
- As `native-image` does not support cross-compilation, you can keep an OS neutral deployment artifact which you convert later to different OS architectures.

You can convert a Spring Boot executable jar into a native image using Cloud Native Buildpacks, or using the `native-image` tool that is shipped with GraalVM.

NOTE	Your executable jar must include AOT generated assets such as generated classes and JSON hint files.

Using Buildpacks

Spring Boot applications usually use Cloud Native Buildpacks through the Maven (`mvn spring-boot:build-image`) or Gradle (`gradle bootBuildImage`) integrations. You can, however, also use `pack` to turn an AOT processed Spring Boot executable jar into a native container image.

First, make sure that a Docker daemon is available (see Get Docker for more details). Configure it to allow non-root user if you are on Linux.

You also need to install `pack` by following the installation guide on buildpacks.io.

Assuming an AOT processed Spring Boot executable jar built as `myproject-0.0.1-SNAPSHOT.jar` is in the `target` directory, run:

```
$ pack build --builder paketobuildpacks/builder-jammy-tiny \
    --path target/myproject-0.0.1-SNAPSHOT.jar \
    --env 'BP_NATIVE_IMAGE=true' \
    my-application:0.0.1-SNAPSHOT
```

NOTE You do not need to have a local GraalVM installation to generate an image in this way.

Once `pack` has finished, you can launch the application using `docker run`:

```
$ docker run --rm -p 8080:8080 docker.io/library/myproject:0.0.1-SNAPSHOT
```

Using GraalVM native-image

Another option to turn an AOT processed Spring Boot executable jar into a native executable is to use the GraalVM `native-image` tool. For this to work, you'll need a GraalVM distribution on your machine. You can either download it manually on the Liberica Native Image Kit page or you can use a download manager like SDKMAN!.

Assuming an AOT processed Spring Boot executable jar built as `myproject-0.0.1-SNAPSHOT.jar` is in the `target` directory, run:

```
$ rm -rf target/native
$ mkdir -p target/native
$ cd target/native
$ jar -xvf ../myproject-0.0.1-SNAPSHOT.jar
$ native-image -H:Name=myproject @META-INF/native-image/argfile -cp .:BOOT-
INF/classes:`find BOOT-INF/lib | tr '\n' ':'`
$ mv myproject ../
```

NOTE These commands work on Linux or macOS machines, but you will need to adapt them for Windows.

TIP The `@META-INF/native-image/argfile` might not be packaged in your jar. It is only included when reachability metadata overrides are needed.

WARNING	The `native-image -cp` flag does not accept wildcards. You need to ensure that all jars are listed (the command above uses `find` and `tr` to do this).

15.4.3. Using the Tracing Agent

The GraalVM native image tracing agent allows you to intercept reflection, resources or proxy usage on the JVM in order to generate the related hints. Spring should generate most of these hints automatically, but the tracing agent can be used to quickly identify the missing entries.

When using the agent to generate hints for a native image, there are a couple of approaches:

- Launch the application directly and exercise it.
- Run application tests to exercise the application.

The first option is interesting for identifying the missing hints when a library or a pattern is not recognized by Spring.

The second option sounds more appealing for a repeatable setup, but by default the generated hints will include anything required by the test infrastructure. Some of these will be unnecessary when the application runs for real. To address this problem the agent supports an access-filter file that will cause certain data to be excluded from the generated output.

Launch the Application Directly

Use the following command to launch the application with the native image tracing agent attached:

```
$ java -Dspring.aot.enabled=true \
    -agentlib:native-image-agent=config-output-dir=/path/to/config-dir/ \
    -jar target/myproject-0.0.1-SNAPSHOT.jar
```

Now you can exercise the code paths you want to have hints for and then stop the application with `ctrl-c`.

On application shutdown the native image tracing agent will write the hint files to the given config output directory. You can either manually inspect these files, or use them as input to the native image build process. To use them as input, copy them into the `src/main/resources/META-INF/native-image/` directory. The next time you build the native image, GraalVM will take these files into consideration.

There are more advanced options which can be set on the native image tracing agent, for example filtering the recorded hints by caller classes, etc. For further reading, please see the official documentation.

15.4.4. Custom Hints

If you need to provide your own hints for reflection, resources, serialization, proxy usage etc. you can use the `RuntimeHintsRegistrar` API. Create a class that implements the `RuntimeHintsRegistrar` interface, and then make appropriate calls to the provided `RuntimeHints` instance:

```
import java.lang.reflect.Method;

import org.springframework.aot.hint.ExecutableMode;
import org.springframework.aot.hint.RuntimeHints;
import org.springframework.aot.hint.RuntimeHintsRegistrar;
import org.springframework.util.ReflectionUtils;

public class MyRuntimeHints implements RuntimeHintsRegistrar {

    @Override
    public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
        // Register method for reflection
        Method method = ReflectionUtils.findMethod(MyClass.class, "sayHello",
String.class);
        hints.reflection().registerMethod(method, ExecutableMode.INVOKE);

        // Register resources
        hints.resources().registerPattern("my-resource.txt");

        // Register serialization
        hints.serialization().registerType(MySerializableClass.class);

        // Register proxy
        hints.proxies().registerJdkProxy(MyInterface.class);
    }

}
```

You can then use `@ImportRuntimeHints` on any `@Configuration` class (for example your `@SpringBootApplication` annotated application class) to activate those hints.

If you have classes which need binding (mostly needed when serializing or deserializing JSON), you can use `@RegisterReflectionForBinding` on any bean. Most of the hints are automatically inferred, for example when accepting or returning data from a `@RestController` method. But when you work with `WebClient` or `RestTemplate` directly, you might need to use `@RegisterReflectionForBinding`.

Testing custom hints

The `RuntimeHintsPredicates` API can be used to test your hints. The API provides methods that build a `Predicate` that can be used to test a `RuntimeHints` instance.

If you're using AssertJ, your test would look like this:

```
import org.junit.jupiter.api.Test;

import org.springframework.aot.hint.RuntimeHints;
import org.springframework.aot.hint.predicate.RuntimeHintsPredicates;
import org.springframework.boot.docs.nativeimage.advanced.customhints.MyRuntimeHints;

import static org.assertj.core.api.Assertions.assertThat;

class MyRuntimeHintsTests {

    @Test
    void shouldRegisterHints() {
        RuntimeHints hints = new RuntimeHints();
        new MyRuntimeHints().registerHints(hints, getClass().getClassLoader());
        assertThat(RuntimeHintsPredicates.resource().forResource("my-
resource.txt")).accepts(hints);
    }

}
```

15.4.5. Known Limitations

GraalVM native images are an evolving technology and not all libraries provide support. The GraalVM community is helping by providing reachability metadata for projects that don't yet ship their own. Spring itself doesn't contain hints for 3rd party libraries and instead relies on the reachability metadata project.

If you encounter problems when generating native images for Spring Boot applications, please check the Spring Boot with GraalVM page of the Spring Boot wiki. You can also contribute issues to the spring-aot-smoke-tests project on GitHub which is used to confirm that common application types are working as expected.

If you find a library which doesn't work with GraalVM, please raise an issue on the reachability metadata project.

15.5. What to Read Next

If you want to learn more about the ahead-of-time processing provided by our build plugins, see the Maven and Gradle plugin documentation. To learn more about the APIs used to perform the processing, browse the `org.springframework.aot.generate` and `org.springframework.beans.factory.aot` packages of the Spring Framework sources.

For known limitations with Spring and GraalVM, please see the Spring Boot wiki.

The next section goes on to cover the *Spring Boot CLI*.

Chapter 16. Spring Boot CLI

The Spring Boot CLI is a command line tool that you can use to bootstrap a new project from start.spring.io or encode a password.

16.1. Installing the CLI

The Spring Boot CLI (Command-Line Interface) can be installed manually by using SDKMAN! (the SDK Manager) or by using Homebrew or MacPorts if you are an OSX user. See *Installing the Spring Boot CLI* in the "Getting started" section for comprehensive installation instructions.

16.2. Using the CLI

Once you have installed the CLI, you can run it by typing `spring` and pressing Enter at the command line. If you run `spring` without any arguments, a help screen is displayed, as follows:

```
$ spring
usage: spring [--help] [--version]
       <command> [<args>]

Available commands are:

  init [options] [location]
    Initialize a new project using Spring Initializr (start.spring.io)

  encodepassword [options] <password to encode>
    Encode a password for use with Spring Security

  shell
    Start a nested shell

Common options:

  --debug Verbose mode
    Print additional status information for the command you are running


See 'spring help <command>' for more information on a specific command.
```

You can type `spring help` to get more details about any of the supported commands, as shown in the following example:

```
$ spring help init
spring init - Initialize a new project using Spring Initializr (start.spring.io)

usage: spring init [options] [location]

Option                        Description
------                        -----------
-a, --artifact-id <String>    Project coordinates; infer archive name (for
                                example 'test')
-b, --boot-version <String>   Spring Boot version (for example '1.2.0.RELEASE')
--build <String>              Build system to use (for example 'maven' or
                                'gradle') (default: maven)
-d, --dependencies <String>   Comma-separated list of dependency identifiers to
                                include in the generated project
--description <String>        Project description
-f, --force                   Force overwrite of existing files
--format <String>             Format of the generated content (for example
                                'build' for a build file, 'project' for a
                                project archive) (default: project)
-g, --group-id <String>       Project coordinates (for example 'org.test')
-j, --java-version <String>   Language level (for example '1.8')
-l, --language <String>       Programming language  (for example 'java')
--list                        List the capabilities of the service. Use it to
                                discover the dependencies and the types that are
                                available
-n, --name <String>           Project name; infer application name
-p, --packaging <String>      Project packaging (for example 'jar')
--package-name <String>       Package name
-t, --type <String>           Project type. Not normally needed if you use --
                                build and/or --format. Check the capabilities of
                                the service (--list) for more details
--target <String>             URL of the service to use (default: https://start.
                                spring.io)
-v, --version <String>        Project version (for example '0.0.1-SNAPSHOT')
-x, --extract                 Extract the project archive. Inferred if a
                                location is specified without an extension

examples:

    To list all the capabilities of the service:
        $ spring init --list

    To creates a default project:
        $ spring init

    To create a web my-app.zip:
        $ spring init -d=web my-app.zip

    To create a web/data-jpa gradle project unpacked:
        $ spring init -d=web,jpa --build=gradle my-dir
```

The `version` command provides a quick way to check which version of Spring Boot you are using, as follows:

```
$ spring version
Spring CLI v3.1.6
```

16.2.1. Initialize a New Project

The `init` command lets you create a new project by using start.spring.io without leaving the shell, as shown in the following example:

```
$ spring init --dependencies=web,data-jpa my-project
Using service at https://start.spring.io
Project extracted to '/Users/developer/example/my-project'
```

The preceding example creates a `my-project` directory with a Maven-based project that uses `spring-boot-starter-web` and `spring-boot-starter-data-jpa`. You can list the capabilities of the service by using the `--list` flag, as shown in the following example:

```
$ spring init --list
=======================================
Capabilities of https://start.spring.io
=======================================

Available dependencies:
-----------------------
actuator - Actuator: Production ready features to help you monitor and manage your
application
...
web - Web: Support for full-stack web development, including Tomcat and spring-webmvc
websocket - Websocket: Support for WebSocket development
ws - WS: Support for Spring Web Services

Available project types:
------------------------
gradle-build -  Gradle Config [format:build, build:gradle]
gradle-project -  Gradle Project [format:project, build:gradle]
maven-build -  Maven POM [format:build, build:maven]
maven-project -  Maven Project [format:project, build:maven] (default)

...
```

The `init` command supports many options. See the `help` output for more details. For instance, the following command creates a Gradle project that uses Java 17 and `war` packaging:

```
$ spring init --build=gradle --java-version=17 --dependencies=websocket
--packaging=war sample-app.zip
Using service at https://start.spring.io
Content saved to 'sample-app.zip'
```

16.2.2. Using the Embedded Shell

Spring Boot includes command-line completion scripts for the BASH and zsh shells. If you do not use either of these shells (perhaps you are a Windows user), you can use the `shell` command to launch an integrated shell, as shown in the following example:

```
$ spring shell
Spring Boot (v3.1.6)
Hit TAB to complete. Type \'help' and hit RETURN for help, and \'exit' to quit.
```

From inside the embedded shell, you can run other commands directly:

```
$ version
Spring CLI v3.1.6
```

The embedded shell supports ANSI color output as well as `tab` completion. If you need to run a native command, you can use the `!` prefix. To exit the embedded shell, press `ctrl-c`.

Chapter 17. Build Tool Plugins

Spring Boot provides build tool plugins for Maven and Gradle. The plugins offer a variety of features, including the packaging of executable jars. This section provides more details on both plugins as well as some help should you need to extend an unsupported build system. If you are just getting started, you might want to read "Build Systems" from the "Developing with Spring Boot" section first.

17.1. Spring Boot Maven Plugin

The Spring Boot Maven Plugin provides Spring Boot support in Maven, letting you package executable jar or war archives and run an application "in-place". To use it, you must use Maven 3.6.3 or later.

See the plugin's documentation to learn more:

- Reference (HTML and PDF)
- API

17.2. Spring Boot Gradle Plugin

The Spring Boot Gradle Plugin provides Spring Boot support in Gradle, letting you package executable jar or war archives, run Spring Boot applications, and use the dependency management provided by `spring-boot-dependencies`. It requires Gradle 7.x (7.5 or later) or 8.x. See the plugin's documentation to learn more:

- Reference (HTML and PDF)
- API

17.3. Spring Boot AntLib Module

The Spring Boot AntLib module provides basic Spring Boot support for Apache Ant. You can use the module to create executable jars. To use the module, you need to declare an additional `spring-boot` namespace in your `build.xml`, as shown in the following example:

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
    xmlns:spring-boot="antlib:org.springframework.boot.ant"
    name="myapp" default="build">
    ...
</project>
```

You need to remember to start Ant using the `-lib` option, as shown in the following example:

```
$ ant -lib <directory containing spring-boot-antlib-3.1.6.jar>
```

TIP	The “Using Spring Boot” section includes a more complete example of using Apache Ant with `spring-boot-antlib`.

17.3.1. Spring Boot Ant Tasks

Once the `spring-boot-antlib` namespace has been declared, the following additional tasks are available:

- Using the “exejar” Task
- Using the “findmainclass” Task

Using the “exejar” Task

You can use the `exejar` task to create a Spring Boot executable jar. The following attributes are supported by the task:

Attribute	Description	Required
`destfile`	The destination jar file to create	Yes
`classes`	The root directory of Java class files	Yes
`start-class`	The main application class to run	No *(the default is the first class found that declares a `main` method)*

The following nested elements can be used with the task:

Element	Description
`resources`	One or more Resource Collections describing a set of Resources that should be added to the content of the created jar file.
`lib`	One or more Resource Collections that should be added to the set of jar libraries that make up the runtime dependency classpath of the application.

Examples

This section shows two examples of Ant tasks.

Specify start-class

```
<spring-boot:exejar destfile="target/my-application.jar"
        classes="target/classes" start-class="com.example.MyApplication">
    <resources>
        <fileset dir="src/main/resources" />
    </resources>
    <lib>
        <fileset dir="lib" />
    </lib>
</spring-boot:exejar>
```

Detect start-class

```
<exejar destfile="target/my-application.jar" classes="target/classes">
    <lib>
        <fileset dir="lib" />
    </lib>
</exejar>
```

17.3.2. Using the "findmainclass" Task

The `findmainclass` task is used internally by `exejar` to locate a class declaring a `main`. If necessary, you can also use this task directly in your build. The following attributes are supported:

Attribute	Description	Required
`classesroot`	The root directory of Java class files	Yes *(unless* `mainclass` *is specified)*
`mainclass`	Can be used to short-circuit the `main` class search	No
`property`	The Ant property that should be set with the result	No *(result will be logged if unspecified)*

Examples

This section contains three examples of using `findmainclass`.

Find and log

```
<findmainclass classesroot="target/classes" />
```

Find and set

```
<findmainclass classesroot="target/classes" property="main-class" />
```

Override and set

```
<findmainclass mainclass="com.example.MainClass" property="main-class" />
```

17.4. Supporting Other Build Systems

If you want to use a build tool other than Maven, Gradle, or Ant, you likely need to develop your own plugin. Executable jars need to follow a specific format and certain entries need to be written in an uncompressed form (see the "executable jar format" section in the appendix for details).

The Spring Boot Maven and Gradle plugins both make use of `spring-boot-loader-tools` to actually generate jars. If you need to, you may use this library directly.

17.4.1. Repackaging Archives

To repackage an existing archive so that it becomes a self-contained executable archive, use `org.springframework.boot.loader.tools.Repackager`. The `Repackager` class takes a single constructor argument that refers to an existing jar or war archive. Use one of the two available `repackage()` methods to either replace the original file or write to a new destination. Various settings can also be configured on the repackager before it is run.

17.4.2. Nested Libraries

When repackaging an archive, you can include references to dependency files by using the `org.springframework.boot.loader.tools.Libraries` interface. We do not provide any concrete implementations of `Libraries` here as they are usually build-system-specific.

If your archive already includes libraries, you can use `Libraries.NONE`.

17.4.3. Finding a Main Class

If you do not use `Repackager.setMainClass()` to specify a main class, the repackager uses ASM to read class files and tries to find a suitable class with a `public static void main(String[] args)` method. An exception is thrown if more than one candidate is found.

17.4.4. Example Repackage Implementation

The following example shows a typical repackage implementation:

Java

```
import java.io.File;
import java.io.IOException;
import java.util.List;

import org.springframework.boot.loader.tools.Library;
import org.springframework.boot.loader.tools.LibraryCallback;
import org.springframework.boot.loader.tools.LibraryScope;
import org.springframework.boot.loader.tools.Repackager;

public class MyBuildTool {

    public void build() throws IOException {
        File sourceJarFile = ...
        Repackager repackager = new Repackager(sourceJarFile);
        repackager.setBackupSource(false);
        repackager.repackage(this::getLibraries);
    }

    private void getLibraries(LibraryCallback callback) throws IOException {
        // Build system specific implementation, callback for each dependency
        for (File nestedJar : getCompileScopeJars()) {
            callback.library(new Library(nestedJar, LibraryScope.COMPILE));
        }
        // ...
    }

    private List<File> getCompileScopeJars() {
        return ...
    }

}
```

Kotlin

```
import org.springframework.boot.loader.tools.Library
import org.springframework.boot.loader.tools.LibraryCallback
import org.springframework.boot.loader.tools.LibraryScope
import org.springframework.boot.loader.tools.Repackager
import java.io.File
import java.io.IOException

class MyBuildTool {

    @Throws(IOException::class)
    fun build() {
        val sourceJarFile: File? =  ...
        val repackager = Repackager(sourceJarFile)
        repackager.setBackupSource(false)
        repackager.repackage { callback: LibraryCallback -> getLibraries(callback) }
    }

    @Throws(IOException::class)
    private fun getLibraries(callback: LibraryCallback) {
        // Build system specific implementation, callback for each dependency
        for (nestedJar in getCompileScopeJars()!!) {
            callback.library(Library(nestedJar, LibraryScope.COMPILE))
        }
        // ...
    }

    private fun getCompileScopeJars(): List<File?>? {
        return  ...
    }

}
```

17.5. What to Read Next

If you are interested in how the build tool plugins work, you can look at the `spring-boot-tools` module on GitHub. More technical details of the executable jar format are covered in the appendix.

If you have specific build-related questions, see the "how-to" guides.

Chapter 18. “How-to” Guides

This section provides answers to some common ‘how do I do that…’ questions that often arise when using Spring Boot. Its coverage is not exhaustive, but it does cover quite a lot.

If you have a specific problem that we do not cover here, you might want to check stackoverflow.com to see if someone has already provided an answer. This is also a great place to ask new questions (please use the `spring-boot` tag).

We are also more than happy to extend this section. If you want to add a ‘how-to’, send us a pull request.

18.1. Spring Boot Application

This section includes topics relating directly to Spring Boot applications.

18.1.1. Create Your Own FailureAnalyzer

`FailureAnalyzer` is a great way to intercept an exception on startup and turn it into a human-readable message, wrapped in a `FailureAnalysis`. Spring Boot provides such an analyzer for application-context-related exceptions, JSR-303 validations, and more. You can also create your own.

`AbstractFailureAnalyzer` is a convenient extension of `FailureAnalyzer` that checks the presence of a specified exception type in the exception to handle. You can extend from that so that your implementation gets a chance to handle the exception only when it is actually present. If, for whatever reason, you cannot handle the exception, return `null` to give another implementation a chance to handle the exception.

`FailureAnalyzer` implementations must be registered in `META-INF/spring.factories`. The following example registers `ProjectConstraintViolationFailureAnalyzer`:

```
org.springframework.boot.diagnostics.FailureAnalyzer=\
com.example.ProjectConstraintViolationFailureAnalyzer
```

NOTE If you need access to the `BeanFactory` or the `Environment`, your `FailureAnalyzer` can implement `BeanFactoryAware` or `EnvironmentAware` respectively.

18.1.2. Troubleshoot Auto-configuration

The Spring Boot auto-configuration tries its best to “do the right thing”, but sometimes things fail, and it can be hard to tell why.

There is a really useful `ConditionEvaluationReport` available in any Spring Boot `ApplicationContext`. You can see it if you enable `DEBUG` logging output. If you use the `spring-boot-actuator` (see the Actuator chapter), there is also a `conditions` endpoint that renders the report in JSON. Use that endpoint to debug the application and see what features have been added (and which have not

been added) by Spring Boot at runtime.

Many more questions can be answered by looking at the source code and the Javadoc. When reading the code, remember the following rules of thumb:

- Look for classes called `*AutoConfiguration` and read their sources. Pay special attention to the `@Conditional*` annotations to find out what features they enable and when. Add `--debug` to the command line or a System property `-Ddebug` to get a log on the console of all the auto-configuration decisions that were made in your app. In a running application with actuator enabled, look at the `conditions` endpoint (`/actuator/conditions` or the JMX equivalent) for the same information.
- Look for classes that are `@ConfigurationProperties` (such as `ServerProperties`) and read from there the available external configuration options. The `@ConfigurationProperties` annotation has a `name` attribute that acts as a prefix to external properties. Thus, `ServerProperties` has `prefix="server"` and its configuration properties are `server.port`, `server.address`, and others. In a running application with actuator enabled, look at the `configprops` endpoint.
- Look for uses of the `bind` method on the `Binder` to pull configuration values explicitly out of the `Environment` in a relaxed manner. It is often used with a prefix.
- Look for `@Value` annotations that bind directly to the `Environment`.
- Look for `@ConditionalOnExpression` annotations that switch features on and off in response to SpEL expressions, normally evaluated with placeholders resolved from the `Environment`.

18.1.3. Customize the Environment or ApplicationContext Before It Starts

A `SpringApplication` has `ApplicationListeners` and `ApplicationContextInitializers` that are used to apply customizations to the context or environment. Spring Boot loads a number of such customizations for use internally from `META-INF/spring.factories`. There is more than one way to register additional customizations:

- Programmatically, per application, by calling the `addListeners` and `addInitializers` methods on `SpringApplication` before you run it.
- Declaratively, per application, by setting the `context.initializer.classes` or `context.listener.classes` properties.
- Declaratively, for all applications, by adding a `META-INF/spring.factories` and packaging a jar file that the applications all use as a library.

The `SpringApplication` sends some special `ApplicationEvents` to the listeners (some even before the context is created) and then registers the listeners for events published by the `ApplicationContext` as well. See "Application Events and Listeners" in the 'Spring Boot features' section for a complete list.

It is also possible to customize the `Environment` before the application context is refreshed by using `EnvironmentPostProcessor`. Each implementation should be registered in `META-INF/spring.factories`, as shown in the following example:

```
org.springframework.boot.env.EnvironmentPostProcessor=com.example.YourEnvironmentPostP
rocessor
```

The implementation can load arbitrary files and add them to the `Environment`. For instance, the following example loads a YAML configuration file from the classpath:

Java

```
import java.io.IOException;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.env.EnvironmentPostProcessor;
import org.springframework.boot.env.YamlPropertySourceLoader;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.PropertySource;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import org.springframework.util.Assert;

public class MyEnvironmentPostProcessor implements EnvironmentPostProcessor {

    private final YamlPropertySourceLoader loader = new YamlPropertySourceLoader();

    @Override
    public void postProcessEnvironment(ConfigurableEnvironment environment,
SpringApplication application) {
        Resource path = new ClassPathResource("com/example/myapp/config.yml");
        PropertySource<?> propertySource = loadYaml(path);
        environment.getPropertySources().addLast(propertySource);
    }

    private PropertySource<?> loadYaml(Resource path) {
        Assert.isTrue(path.exists(), () -> "Resource " + path + " does not exist");
        try {
            return this.loader.load("custom-resource", path).get(0);
        }
        catch (IOException ex) {
            throw new IllegalStateException("Failed to load yaml configuration from "
+ path, ex);
        }
    }

}
```

Kotlin

```
import org.springframework.boot.SpringApplication
import org.springframework.boot.env.EnvironmentPostProcessor
import org.springframework.boot.env.YamlPropertySourceLoader
import org.springframework.core.env.ConfigurableEnvironment
import org.springframework.core.env.PropertySource
import org.springframework.core.io.ClassPathResource
import org.springframework.core.io.Resource
import org.springframework.util.Assert
import java.io.IOException

class MyEnvironmentPostProcessor : EnvironmentPostProcessor {

    private val loader = YamlPropertySourceLoader()

    override fun postProcessEnvironment(environment: ConfigurableEnvironment,
application: SpringApplication) {
        val path: Resource = ClassPathResource("com/example/myapp/config.yml")
        val propertySource = loadYaml(path)
        environment.propertySources.addLast(propertySource)
    }

    private fun loadYaml(path: Resource): PropertySource<*> {
        Assert.isTrue(path.exists()) { "Resource $path does not exist" }
        return try {
            loader.load("custom-resource", path)[0]
        } catch (ex: IOException) {
            throw IllegalStateException("Failed to load yaml configuration from
$path", ex)
        }
    }

}
```

TIP The `Environment` has already been prepared with all the usual property sources that Spring Boot loads by default. It is therefore possible to get the location of the file from the environment. The preceding example adds the `custom-resource` property source at the end of the list so that a key defined in any of the usual other locations takes precedence. A custom implementation may define another order.

CAUTION While using `@PropertySource` on your `@SpringBootApplication` may seem to be a convenient way to load a custom resource in the `Environment`, we do not recommend it. Such property sources are not added to the `Environment` until the application context is being refreshed. This is too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.

18.1.4. Build an ApplicationContext Hierarchy (Adding a Parent or Root Context)

You can use the `ApplicationBuilder` class to create parent/child `ApplicationContext` hierarchies. See "Fluent Builder API" in the 'Spring Boot features' section for more information.

18.1.5. Create a Non-web Application

Not all Spring applications have to be web applications (or web services). If you want to execute some code in a `main` method but also bootstrap a Spring application to set up the infrastructure to use, you can use the `SpringApplication` features of Spring Boot. A `SpringApplication` changes its `ApplicationContext` class, depending on whether it thinks it needs a web application or not. The first thing you can do to help it is to leave server-related dependencies (such as the servlet API) off the classpath. If you cannot do that (for example, you run two applications from the same code base) then you can explicitly call `setWebApplicationType(WebApplicationType.NONE)` on your `SpringApplication` instance or set the `applicationContextClass` property (through the Java API or with external properties). Application code that you want to run as your business logic can be implemented as a `CommandLineRunner` and dropped into the context as a `@Bean` definition.

18.2. Properties and Configuration

This section includes topics about setting and reading properties and configuration settings and their interaction with Spring Boot applications.

18.2.1. Automatically Expand Properties at Build Time

Rather than hardcoding some properties that are also specified in your project's build configuration, you can automatically expand them by instead using the existing build configuration. This is possible in both Maven and Gradle.

Automatic Property Expansion Using Maven

You can automatically expand properties from the Maven project by using resource filtering. If you use the `spring-boot-starter-parent`, you can then refer to your Maven 'project properties' with `@..@` placeholders, as shown in the following example:

Properties

```
app.encoding=@project.build.sourceEncoding@
app.java.version=@java.version@
```

Yaml

```
app:
  encoding: "@project.build.sourceEncoding@"
  java:
    version: "@java.version@"
```

NOTE	Only production configuration is filtered that way (in other words, no filtering is applied on `src/test/resources`).
TIP	If you enable the `addResources` flag, the `spring-boot:run` goal can add `src/main/resources` directly to the classpath (for hot reloading purposes). Doing so circumvents the resource filtering and this feature. Instead, you can use the `exec:java` goal or customize the plugin's configuration. See the plugin usage page for more details.

If you do not use the starter parent, you need to include the following element inside the `<build/>` element of your `pom.xml`:

```
<resources>
    <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
    </resource>
</resources>
```

You also need to include the following element inside `<plugins/>`:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-resources-plugin</artifactId>
    <version>2.7</version>
    <configuration>
        <delimiters>
            <delimiter>@</delimiter>
        </delimiters>
        <useDefaultDelimiters>false</useDefaultDelimiters>
    </configuration>
</plugin>
```

NOTE	The `useDefaultDelimiters` property is important if you use standard Spring placeholders (such as `${placeholder}`) in your configuration. If that property is not set to `false`, these may be expanded by the build.

Automatic Property Expansion Using Gradle

You can automatically expand properties from the Gradle project by configuring the Java plugin's `processResources` task to do so, as shown in the following example:

```
tasks.named('processResources') {
    expand(project.properties)
}
```

You can then refer to your Gradle project's properties by using placeholders, as shown in the following example:

Properties

```
app.name=${name}
app.description=${description}
```

Yaml

```
app:
  name: "${name}"
  description: "${description}"
```

NOTE	Gradle's `expand` method uses Groovy's `SimpleTemplateEngine`, which transforms `${..}` tokens. The `${..}` style conflicts with Spring's own property placeholder mechanism. To use Spring property placeholders together with automatic expansion, escape the Spring property placeholders as follows: `\${..}`.

18.2.2. Externalize the Configuration of SpringApplication

A `SpringApplication` has bean property setters, so you can use its Java API as you create the application to modify its behavior. Alternatively, you can externalize the configuration by setting properties in `spring.main.*`. For example, in `application.properties`, you might have the following settings:

Properties

```
spring.main.web-application-type=none
spring.main.banner-mode=off
```

Yaml

```
spring:
  main:
    web-application-type: "none"
    banner-mode: "off"
```

Then the Spring Boot banner is not printed on startup, and the application is not starting an embedded web server.

Properties defined in external configuration override and replace the values specified with the Java API, with the notable exception of the primary sources. Primary sources are those provided to the `SpringApplication` constructor:

Java

```
import org.springframework.boot.Banner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(MyApplication.class);
        application.setBannerMode(Banner.Mode.OFF);
        application.run(args);
    }

}
```

Kotlin

```
import org.springframework.boot.Banner
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication

@SpringBootApplication
object MyApplication {

    @JvmStatic
    fun main(args: Array<String>) {
        val application = SpringApplication(MyApplication::class.java)
        application.setBannerMode(Banner.Mode.OFF)
        application.run(*args)
    }

}
```

Or to `sources(…)` method of a `SpringApplicationBuilder`:

Java

```
import org.springframework.boot.Banner;
import org.springframework.boot.builder.SpringApplicationBuilder;

public class MyApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder()
            .bannerMode(Banner.Mode.OFF)
            .sources(MyApplication.class)
            .run(args);
    }

}
```

Kotlin

```
import org.springframework.boot.Banner
import org.springframework.boot.builder.SpringApplicationBuilder

object MyApplication {

    @JvmStatic
    fun main(args: Array<String>) {
        SpringApplicationBuilder()
            .bannerMode(Banner.Mode.OFF)
            .sources(MyApplication::class.java)
            .run(*args)
    }

}
```

Given the examples above, if we have the following configuration:

Properties

```
spring.main.sources=com.example.MyDatabaseConfig,com.example.MyJmsConfig
spring.main.banner-mode=console
```

Yaml

```
spring:
  main:
    sources: "com.example.MyDatabaseConfig,com.example.MyJmsConfig"
    banner-mode: "console"
```

The actual application will show the banner (as overridden by configuration) and uses three sources for the `ApplicationContext`. The application sources are:

1. `MyApplication` (from the code)
2. `MyDatabaseConfig` (from the external config)
3. `MyJmsConfig`(from the external config)

18.2.3. Change the Location of External Properties of an Application

By default, properties from different sources are added to the Spring `Environment` in a defined order (see "Externalized Configuration" in the 'Spring Boot features' section for the exact order).

You can also provide the following System properties (or environment variables) to change the behavior:

- `spring.config.name` (`SPRING_CONFIG_NAME`): Defaults to `application` as the root of the file name.
- `spring.config.location` (`SPRING_CONFIG_LOCATION`): The file to load (such as a classpath resource or a URL). A separate `Environment` property source is set up for this document and it can be overridden by system properties, environment variables, or the command line.

No matter what you set in the environment, Spring Boot always loads `application.properties` as described above. By default, if YAML is used, then files with the '.yaml' and '.yml' extension are also added to the list.

TIP If you want detailed information about the files that are being loaded you can set the logging level of `org.springframework.boot.context.config` to `trace`.

18.2.4. Use 'Short' Command Line Arguments

Some people like to use (for example) `--port=9000` instead of `--server.port=9000` to set configuration properties on the command line. You can enable this behavior by using placeholders in `application.properties`, as shown in the following example:

Properties

```
server.port=${port:8080}
```

Yaml

```
server:
  port: "${port:8080}"
```

TIP If you inherit from the `spring-boot-starter-parent` POM, the default filter token of the `maven-resources-plugins` has been changed from `${*}` to `@` (that is, `@maven.token@` instead of `${maven.token}`) to prevent conflicts with Spring-style placeholders. If you have enabled Maven filtering for the `application.properties` directly, you may want to also change the default filter token to use other delimiters.

NOTE	In this specific case, the port binding works in a PaaS environment such as Heroku or Cloud Foundry. In those two platforms, the `PORT` environment variable is set automatically and Spring can bind to capitalized synonyms for `Environment` properties.

18.2.5. Use YAML for External Properties

YAML is a superset of JSON and, as such, is a convenient syntax for storing external properties in a hierarchical format, as shown in the following example:

```
spring:
  application:
    name: "cruncher"
  datasource:
    driver-class-name: "com.mysql.jdbc.Driver"
    url: "jdbc:mysql://localhost/test"
server:
  port: 9000
```

Create a file called `application.yaml` and put it in the root of your classpath. Then add `snakeyaml` to your dependencies (Maven coordinates `org.yaml:snakeyaml`, already included if you use the `spring-boot-starter`). A YAML file is parsed to a Java `Map<String,Object>` (like a JSON object), and Spring Boot flattens the map so that it is one level deep and has period-separated keys, as many people are used to with `Properties` files in Java.

The preceding example YAML corresponds to the following `application.properties` file:

```
spring.application.name=cruncher
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

See “Working With YAML” in the ‘Spring Boot features’ section for more information about YAML.

18.2.6. Set the Active Spring Profiles

The Spring `Environment` has an API for this, but you would normally set a System property (`spring.profiles.active`) or an OS environment variable (`SPRING_PROFILES_ACTIVE`). Also, you can launch your application with a `-D` argument (remember to put it before the main class or jar archive), as follows:

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

In Spring Boot, you can also set the active profile in `application.properties`, as shown in the following example:

Properties

```
spring.profiles.active=production
```

Yaml

```
spring:
  profiles:
    active: "production"
```

A value set this way is replaced by the System property or environment variable setting but not by the `SpringApplicationBuilder.profiles()` method. Thus, the latter Java API can be used to augment the profiles without changing the defaults.

See "Profiles" in the "Spring Boot features" section for more information.

18.2.7. Set the Default Profile Name

The default profile is a profile that is enabled if no profile is active. By default, the name of the default profile is `default`, but it could be changed using a System property (`spring.profiles.default`) or an OS environment variable (`SPRING_PROFILES_DEFAULT`).

In Spring Boot, you can also set the default profile name in `application.properties`, as shown in the following example:

Properties

```
spring.profiles.default=dev
```

Yaml

```
spring:
  profiles:
    default: "dev"
```

See "Profiles" in the "Spring Boot features" section for more information.

18.2.8. Change Configuration Depending on the Environment

Spring Boot supports multi-document YAML and Properties files (see Working With Multi-Document Files for details) which can be activated conditionally based on the active profiles.

If a document contains a `spring.config.activate.on-profile` key, then the profiles value (a comma-separated list of profiles or a profile expression) is fed into the Spring `Environment.acceptsProfiles()` method. If the profile expression matches then that document is included in the final merge (otherwise, it is not), as shown in the following example:

Properties

```
server.port=9000
#---
spring.config.activate.on-profile=development
server.port=9001
#---
spring.config.activate.on-profile=production
server.port=0
```

Yaml

```
server:
  port: 9000
---
spring:
  config:
    activate:
      on-profile: "development"
server:
  port: 9001
---
spring:
  config:
    activate:
      on-profile: "production"
server:
  port: 0
```

In the preceding example, the default port is 9000. However, if the Spring profile called 'development' is active, then the port is 9001. If 'production' is active, then the port is 0.

NOTE	The documents are merged in the order in which they are encountered. Later values override earlier values.

18.2.9. Discover Built-in Options for External Properties

Spring Boot binds external properties from `application.properties` (or YAML files and other places) into an application at runtime. There is not (and technically cannot be) an exhaustive list of all supported properties in a single location, because contributions can come from additional jar files on your classpath.

A running application with the Actuator features has a `configprops` endpoint that shows all the bound and bindable properties available through `@ConfigurationProperties`.

The appendix includes an `application.properties` example with a list of the most common properties supported by Spring Boot. The definitive list comes from searching the source code for `@ConfigurationProperties` and `@Value` annotations as well as the occasional use of `Binder`. For more about the exact ordering of loading properties, see "Externalized Configuration".

18.3. Embedded Web Servers

Each Spring Boot web application includes an embedded web server. This feature leads to a number of how-to questions, including how to change the embedded server and how to configure the embedded server. This section answers those questions.

18.3.1. Use Another Web Server

Many Spring Boot starters include default embedded containers.

- For servlet stack applications, the `spring-boot-starter-web` includes Tomcat by including `spring-boot-starter-tomcat`, but you can use `spring-boot-starter-jetty` or `spring-boot-starter-undertow` instead.
- For reactive stack applications, the `spring-boot-starter-webflux` includes Reactor Netty by including `spring-boot-starter-reactor-netty`, but you can use `spring-boot-starter-tomcat`, `spring-boot-starter-jetty`, or `spring-boot-starter-undertow` instead.

When switching to a different HTTP server, you need to swap the default dependencies for those that you need instead. To help with this process, Spring Boot provides a separate starter for each of the supported HTTP servers.

The following Maven example shows how to exclude Tomcat and include Jetty for Spring MVC:

```
<properties>
    <jakarta-servlet.version>5.0.0</jakarta-servlet.version>
</properties>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <!-- Exclude the Tomcat dependency -->
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<!-- Use Jetty instead -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

NOTE The version of the Jakarta Servlet API has been overridden as, unlike Tomcat 10 and Undertow 2.3, Jetty 11 does not support Servlet 6.0.

WARNING

Downgrading the Servlet API to 5.0 breaks Spring Framework's Servlet-related mocks!

As Jetty needs the Servlet API 5.0, this leaves you with two working arrangements:

- Tests use the Servlet 5.0 API and avoid using Framework's Servlet mocks by only using a full-blown web environment
- Tests use the Servlet 6.0 API and avoid starting Jetty by only using a mock web environment

If a mixture of web environments is required by your application's tests, your test setup may require some structural changes to strictly separate the two web environments.

The following Gradle example configures the necessary dependencies and a module replacement to use Undertow in place of Reactor Netty for Spring WebFlux:

```
dependencies {
    implementation "org.springframework.boot:spring-boot-starter-undertow"
    implementation "org.springframework.boot:spring-boot-starter-webflux"
    modules {
        module("org.springframework.boot:spring-boot-starter-reactor-netty") {
            replacedBy("org.springframework.boot:spring-boot-starter-undertow", "Use
Undertow instead of Reactor Netty")
        }
    }
}
```

NOTE

`spring-boot-starter-reactor-netty` is required to use the `WebClient` class, so you may need to keep a dependency on Netty even when you need to include a different HTTP server.

18.3.2. Disabling the Web Server

If your classpath contains the necessary bits to start a web server, Spring Boot will automatically start it. To disable this behavior configure the `WebApplicationType` in your `application.properties`, as shown in the following example:

Properties

```
spring.main.web-application-type=none
```

Yaml

```
spring:
  main:
    web-application-type: "none"
```

18.3.3. Change the HTTP Port

In a standalone application, the main HTTP port defaults to `8080` but can be set with `server.port` (for example, in `application.properties` or as a System property). Thanks to relaxed binding of `Environment` values, you can also use `SERVER_PORT` (for example, as an OS environment variable).

To switch off the HTTP endpoints completely but still create a `WebApplicationContext`, use `server.port=-1` (doing so is sometimes useful for testing).

For more details, see "Customizing Embedded Servlet Containers" in the 'Spring Boot Features' section, or the `ServerProperties` source code.

18.3.4. Use a Random Unassigned HTTP Port

To scan for a free port (using OS natives to prevent clashes) use `server.port=0`.

18.3.5. Discover the HTTP Port at Runtime

You can access the port the server is running on from log output or from the `WebServerApplicationContext` through its `WebServer`. The best way to get that and be sure it has been initialized is to add a `@Bean` of type `ApplicationListener<WebServerInitializedEvent>` and pull the container out of the event when it is published.

Tests that use `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)` can also inject the actual port into a field by using the `@LocalServerPort` annotation, as shown in the following example:

Java

```
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.server.LocalServerPort;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyWebIntegrationTests {

    @LocalServerPort
    int port;

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.boot.test.web.server.LocalServerPort

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyWebIntegrationTests {

    @LocalServerPort
    var port = 0

    // ...

}
```

NOTE	`@LocalServerPort` is a meta-annotation for `@Value("${local.server.port}")`. Do not try to inject the port in a regular application. As we just saw, the value is set only after the container has been initialized. Contrary to a test, application code callbacks are processed early (before the value is actually available).

18.3.6. Enable HTTP Response Compression

HTTP response compression is supported by Jetty, Tomcat, Reactor Netty, and Undertow. It can be enabled in `application.properties`, as follows:

Properties

```
server.compression.enabled=true
```

Yaml

```
server:
  compression:
    enabled: true
```

By default, responses must be at least 2048 bytes in length for compression to be performed. You can configure this behavior by setting the `server.compression.min-response-size` property.

By default, responses are compressed only if their content type is one of the following:

- `text/html`
- `text/xml`
- `text/plain`
- `text/css`
- `text/javascript`

- `application/javascript`
- `application/json`
- `application/xml`

You can configure this behavior by setting the `server.compression.mime-types` property.

18.3.7. Configure SSL

SSL can be configured declaratively by setting the various `server.ssl.*` properties, typically in `application.properties` or `application.yaml`. The following example shows setting SSL properties using a Java KeyStore file:

Properties

```
server.port=8443
server.ssl.key-store=classpath:keystore.jks
server.ssl.key-store-password=secret
server.ssl.key-password=another-secret
```

Yaml

```
server:
  port: 8443
  ssl:
    key-store: "classpath:keystore.jks"
    key-store-password: "secret"
    key-password: "another-secret"
```

Using configuration such as the preceding example means the application no longer supports a plain HTTP connector at port 8080. Spring Boot does not support the configuration of both an HTTP connector and an HTTPS connector through `application.properties`. If you want to have both, you need to configure one of them programmatically. We recommend using `application.properties` to configure HTTPS, as the HTTP connector is the easier of the two to configure programmatically.

Using PEM-encoded files

You can use PEM-encoded files instead of Java KeyStore files. You should use PKCS#8 key files wherever possible. PEM-encoded PKCS#8 key files start with a `-----BEGIN PRIVATE KEY-----` or `-----BEGIN ENCRYPTED PRIVATE KEY-----` header.

If you have files in other formats, e.g., PKCS#1 (`-----BEGIN RSA PRIVATE KEY-----`) or SEC 1 (`-----BEGIN EC PRIVATE KEY-----`), you can convert them to PKCS#8 using OpenSSL:

```
openssl pkcs8 -topk8 -nocrypt -in <input file> -out <output file>
```

The following example shows setting SSL properties using PEM-encoded certificate and private key files:

Properties

```
server.port=8443
server.ssl.certificate=classpath:my-cert.crt
server.ssl.certificate-private-key=classpath:my-cert.key
server.ssl.trust-certificate=classpath:ca-cert.crt
```

Yaml

```
server:
  port: 8443
  ssl:
    certificate: "classpath:my-cert.crt"
    certificate-private-key: "classpath:my-cert.key"
    trust-certificate: "classpath:ca-cert.crt"
```

Alternatively, the SSL trust material can be configured in an SSL bundle and applied to the web server as shown in this example:

Properties

```
server.port=8443
server.ssl.bundle=example
```

Yaml

```
server:
  port: 8443
  ssl:
    bundle: "example"
```

See `Ssl` for details of all of the supported properties.

18.3.8. Configure HTTP/2

You can enable HTTP/2 support in your Spring Boot application with the `server.http2.enabled` configuration property. Both `h2` (HTTP/2 over TLS) and `h2c` (HTTP/2 over TCP) are supported. To use `h2`, SSL must also be enabled. When SSL is not enabled, `h2c` will be used. You may, for example, want to use `h2c` when your application is running behind a proxy server that is performing TLS termination.

HTTP/2 With Tomcat

Spring Boot ships by default with Tomcat 10.1.x which supports `h2c` and `h2` out of the box. Alternatively, you can use `libtcnative` for `h2` support if the library and its dependencies are installed on the host operating system.

The library directory must be made available, if not already, to the JVM library path. You can do so

with a JVM argument such as `-Djava.library.path=/usr/local/opt/tomcat-native/lib`. More on this in the official Tomcat documentation.

HTTP/2 With Jetty

For HTTP/2 support, Jetty requires the additional `org.eclipse.jetty.http2:http2-server` dependency. To use `h2c` no other dependencies are required. To use `h2`, you also need to choose one of the following dependencies, depending on your deployment:

- `org.eclipse.jetty:jetty-alpn-java-server` to use the JDK built-in support
- `org.eclipse.jetty:jetty-alpn-conscrypt-server` and the Conscrypt library

HTTP/2 With Reactor Netty

The `spring-boot-webflux-starter` is using by default Reactor Netty as a server. Reactor Netty supports `h2c` and `h2` out of the box. For optimal runtime performance, this server also supports `h2` with native libraries. To enable that, your application needs to have an additional dependency.

Spring Boot manages the version for the `io.netty:netty-tcnative-boringssl-static` "uber jar", containing native libraries for all platforms. Developers can choose to import only the required dependencies using a classifier (see the Netty official documentation).

HTTP/2 With Undertow

Undertow supports `h2c` and `h2` out of the box.

18.3.9. Configure the Web Server

Generally, you should first consider using one of the many available configuration keys and customize your web server by adding new entries in your `application.properties` or `application.yaml` file. See "Discover Built-in Options for External Properties"). The `server.*` namespace is quite useful here, and it includes namespaces like `server.tomcat.*`, `server.jetty.*` and others, for server-specific features. See the list of Common Application Properties.

The previous sections covered already many common use cases, such as compression, SSL or HTTP/2. However, if a configuration key does not exist for your use case, you should then look at `WebServerFactoryCustomizer`. You can declare such a component and get access to the server factory relevant to your choice: you should select the variant for the chosen Server (Tomcat, Jetty, Reactor Netty, Undertow) and the chosen web stack (servlet or reactive).

The example below is for Tomcat with the `spring-boot-starter-web` (servlet stack):

Java

```
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyTomcatWebServerCustomizer implements
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    @Override
    public void customize(TomcatServletWebServerFactory factory) {
        // customize the factory here
    }

}
```

Kotlin

```
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.stereotype.Component

@Component
class MyTomcatWebServerCustomizer :
WebServerFactoryCustomizer<TomcatServletWebServerFactory?> {

    override fun customize(factory: TomcatServletWebServerFactory?) {
        // customize the factory here
    }

}
```

NOTE Spring Boot uses that infrastructure internally to auto-configure the server. Auto-configured `WebServerFactoryCustomizer` beans have an order of `0` and will be processed before any user-defined customizers, unless it has an explicit order that states otherwise.

Once you have got access to a `WebServerFactory` using the customizer, you can use it to configure specific parts, like connectors, server resources, or the server itself - all using server-specific APIs.

In addition Spring Boot provides:

Server	Servlet stack	Reactive stack
Tomcat	`TomcatServletWebServerFactory`	`TomcatReactiveWebServerFactory`
Jetty	`JettyServletWebServerFactory`	`JettyReactiveWebServerFactory`
Undertow	`UndertowServletWebServerFactory`	`UndertowReactiveWebServerFactory`

Server	Servlet stack	Reactive stack
Reactor	N/A	`NettyReactiveWebServerFactory`

As a last resort, you can also declare your own `WebServerFactory` bean, which will override the one provided by Spring Boot. When you do so, auto-configured customizers are still applied on your custom factory, so use that option carefully.

18.3.10. Add a Servlet, Filter, or Listener to an Application

In a servlet stack application, that is with the `spring-boot-starter-web`, there are two ways to add `Servlet`, `Filter`, `ServletContextListener`, and the other listeners supported by the Servlet API to your application:

- Add a Servlet, Filter, or Listener by Using a Spring Bean
- Add Servlets, Filters, and Listeners by Using Classpath Scanning

Add a Servlet, Filter, or Listener by Using a Spring Bean

To add a `Servlet`, `Filter`, or servlet `*Listener` by using a Spring bean, you must provide a `@Bean` definition for it. Doing so can be very useful when you want to inject configuration or dependencies. However, you must be very careful that they do not cause eager initialization of too many other beans, because they have to be installed in the container very early in the application lifecycle. (For example, it is not a good idea to have them depend on your `DataSource` or JPA configuration.) You can work around such restrictions by initializing the beans lazily when first used instead of on initialization.

In the case of filters and servlets, you can also add mappings and init parameters by adding a `FilterRegistrationBean` or a `ServletRegistrationBean` instead of or in addition to the underlying component.

NOTE	If no `dispatcherType` is specified on a filter registration, `REQUEST` is used. This aligns with the servlet specification's default dispatcher type.

Like any other Spring bean, you can define the order of servlet filter beans; please make sure to check the "Registering Servlets, Filters, and Listeners as Spring Beans" section.

Disable Registration of a Servlet or Filter

As described earlier, any `Servlet` or `Filter` beans are registered with the servlet container automatically. To disable registration of a particular `Filter` or `Servlet` bean, create a registration bean for it and mark it as disabled, as shown in the following example:

Java

```
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyFilterConfiguration {

    @Bean
    public FilterRegistrationBean<MyFilter> registration(MyFilter filter) {
        FilterRegistrationBean<MyFilter> registration = new
FilterRegistrationBean<>(filter);
        registration.setEnabled(false);
        return registration;
    }

}
```

Kotlin

```
import org.springframework.boot.web.servlet.FilterRegistrationBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyFilterConfiguration {

    @Bean
    fun registration(filter: MyFilter): FilterRegistrationBean<MyFilter> {
        val registration = FilterRegistrationBean(filter)
        registration.isEnabled = false
        return registration
    }

}
```

Add Servlets, Filters, and Listeners by Using Classpath Scanning

`@WebServlet`, `@WebFilter`, and `@WebListener` annotated classes can be automatically registered with an embedded servlet container by annotating a `@Configuration` class with `@ServletComponentScan` and specifying the package(s) containing the components that you want to register. By default, `@ServletComponentScan` scans from the package of the annotated class.

18.3.11. Configure Access Logging

Access logs can be configured for Tomcat, Undertow, and Jetty through their respective namespaces.

For instance, the following settings log access on Tomcat with a custom pattern.

Properties

```
server.tomcat.basedir=my-tomcat
server.tomcat.accesslog.enabled=true
server.tomcat.accesslog.pattern=%t %a %r %s (%D microseconds)
```

Yaml

```
server:
  tomcat:
    basedir: "my-tomcat"
    accesslog:
      enabled: true
      pattern: "%t %a %r %s (%D microseconds)"
```

NOTE The default location for logs is a `logs` directory relative to the Tomcat base directory. By default, the `logs` directory is a temporary directory, so you may want to fix Tomcat's base directory or use an absolute path for the logs. In the preceding example, the logs are available in `my-tomcat/logs` relative to the working directory of the application.

Access logging for Undertow can be configured in a similar fashion, as shown in the following example:

Properties

```
server.undertow.accesslog.enabled=true
server.undertow.accesslog.pattern=%t %a %r %s (%D milliseconds)
server.undertow.options.server.record-request-start-time=true
```

Yaml

```
server:
  undertow:
    accesslog:
      enabled: true
      pattern: "%t %a %r %s (%D milliseconds)"
    options:
      server:
        record-request-start-time: true
```

Note that, in addition to enabling access logging and configuring its pattern, recording request start times has also been enabled. This is required when including the response time (`%D`) in the access log pattern. Logs are stored in a `logs` directory relative to the working directory of the application. You can customize this location by setting the `server.undertow.accesslog.dir` property.

Finally, access logging for Jetty can also be configured as follows:

Properties

```
server.jetty.accesslog.enabled=true
server.jetty.accesslog.filename=/var/log/jetty-access.log
```

Yaml

```
server:
  jetty:
    accesslog:
      enabled: true
      filename: "/var/log/jetty-access.log"
```

By default, logs are redirected to `System.err`. For more details, see the Jetty documentation.

18.3.12. Running Behind a Front-end Proxy Server

If your application is running behind a proxy, a load-balancer or in the cloud, the request information (like the host, port, scheme...) might change along the way. Your application may be running on `10.10.10.10:8080`, but HTTP clients should only see `example.org`.

RFC7239 "Forwarded Headers" defines the `Forwarded` HTTP header; proxies can use this header to provide information about the original request. You can configure your application to read those headers and automatically use that information when creating links and sending them to clients in HTTP 302 responses, JSON documents or HTML pages. There are also non-standard headers, like `X-Forwarded-Host`, `X-Forwarded-Port`, `X-Forwarded-Proto`, `X-Forwarded-Ssl`, and `X-Forwarded-Prefix`.

If the proxy adds the commonly used `X-Forwarded-For` and `X-Forwarded-Proto` headers, setting `server.forward-headers-strategy` to `NATIVE` is enough to support those. With this option, the Web servers themselves natively support this feature; you can check their specific documentation to learn about specific behavior.

If this is not enough, Spring Framework provides a ForwardedHeaderFilter for the servlet stack and a ForwardedHeaderTransformer for the reactive stack. You can use them in your application by setting `server.forward-headers-strategy` to `FRAMEWORK`.

TIP If you are using Tomcat and terminating SSL at the proxy, `server.tomcat.redirect-context-root` should be set to `false`. This allows the `X-Forwarded-Proto` header to be honored before any redirects are performed.

NOTE If your application runs in Cloud Foundry, Heroku or Kubernetes, the `server.forward-headers-strategy` property defaults to `NATIVE`. In all other instances, it defaults to `NONE`.

Customize Tomcat's Proxy Configuration

If you use Tomcat, you can additionally configure the names of the headers used to carry "forwarded" information, as shown in the following example:

Properties

```
server.tomcat.remoteip.remote-ip-header=x-your-remote-ip-header
server.tomcat.remoteip.protocol-header=x-your-protocol-header
```

Yaml

```
server:
  tomcat:
    remoteip:
      remote-ip-header: "x-your-remote-ip-header"
      protocol-header: "x-your-protocol-header"
```

Tomcat is also configured with a regular expression that matches internal proxies that are to be trusted. See the `server.tomcat.remoteip.internal-proxies` entry in the appendix for its default value. You can customize the valve's configuration by adding an entry to `application.properties`, as shown in the following example:

Properties

```
server.tomcat.remoteip.internal-proxies=192\\.168\\.\\d{1,3}\\.\\d{1,3}
```

Yaml

```
server:
  tomcat:
    remoteip:
      internal-proxies: "192\\.168\\.\\d{1,3}\\.\\d{1,3}"
```

NOTE You can trust all proxies by setting the `internal-proxies` to empty (but do not do so in production).

You can take complete control of the configuration of Tomcat's `RemoteIpValve` by switching the automatic one off (to do so, set `server.forward-headers-strategy=NONE`) and adding a new valve instance using a `WebServerFactoryCustomizer` bean.

18.3.13. Enable Multiple Connectors with Tomcat

You can add an `org.apache.catalina.connector.Connector` to the `TomcatServletWebServerFactory`, which can allow multiple connectors, including HTTP and HTTPS connectors, as shown in the following example:

Java

```
import org.apache.catalina.connector.Connector;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyTomcatConfiguration {

    @Bean
    public WebServerFactoryCustomizer<TomcatServletWebServerFactory>
connectorCustomizer() {
        return (tomcat) -> tomcat.addAdditionalTomcatConnectors(createConnector());
    }

    private Connector createConnector() {
        Connector connector = new
Connector("org.apache.coyote.http11.Http11NioProtocol");
        connector.setPort(8081);
        return connector;
    }

}
```

Kotlin

```
import org.apache.catalina.connector.Connector
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyTomcatConfiguration {

    @Bean
    fun connectorCustomizer():
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {
        return WebServerFactoryCustomizer { tomcat: TomcatServletWebServerFactory ->
            tomcat.addAdditionalTomcatConnectors(
                createConnector()
            )
        }
    }

    private fun createConnector(): Connector {
        val connector = Connector("org.apache.coyote.http11.Http11NioProtocol")
        connector.port = 8081
        return connector
    }

}
```

18.3.14. Enable Tomcat's MBean Registry

Embedded Tomcat's MBean registry is disabled by default. This minimizes Tomcat's memory footprint. If you want to use Tomcat's MBeans, for example so that they can be used by Micrometer to expose metrics, you must use the `server.tomcat.mbeanregistry.enabled` property to do so, as shown in the following example:

Properties

```
server.tomcat.mbeanregistry.enabled=true
```

Yaml

```
server:
  tomcat:
    mbeanregistry:
      enabled: true
```

18.3.15. Enable Multiple Listeners with Undertow

Add an `UndertowBuilderCustomizer` to the `UndertowServletWebServerFactory` and add a listener to the `Builder`, as shown in the following example:

Java

```
import io.undertow.Undertow.Builder;

import org.springframework.boot.web.embedded.undertow.UndertowServletWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyUndertowConfiguration {

    @Bean
    public WebServerFactoryCustomizer<UndertowServletWebServerFactory>
undertowListenerCustomizer() {
        return (factory) -> factory.addBuilderCustomizers(this::addHttpListener);
    }

    private Builder addHttpListener(Builder builder) {
        return builder.addHttpListener(8080, "0.0.0.0");
    }

}
```

Kotlin

```
import io.undertow.Undertow
import org.springframework.boot.web.embedded.undertow.UndertowBuilderCustomizer
import org.springframework.boot.web.embedded.undertow.UndertowServletWebServerFactory
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyUndertowConfiguration {

    @Bean
    fun undertowListenerCustomizer():
WebServerFactoryCustomizer<UndertowServletWebServerFactory> {
        return WebServerFactoryCustomizer { factory: UndertowServletWebServerFactory
->
            factory.addBuilderCustomizers(
                UndertowBuilderCustomizer { builder: Undertow.Builder ->
addHttpListener(builder) })
        }
    }

    private fun addHttpListener(builder: Undertow.Builder): Undertow.Builder {
        return builder.addHttpListener(8080, "0.0.0.0")
    }

}
```

18.3.16. Create WebSocket Endpoints Using @ServerEndpoint

If you want to use `@ServerEndpoint` in a Spring Boot application that used an embedded container, you must declare a single `ServerEndpointExporter` `@Bean`, as shown in the following example:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.server.standard.ServerEndpointExporter;

@Configuration(proxyBeanMethods = false)
public class MyWebSocketConfiguration {

    @Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.web.socket.server.standard.ServerEndpointExporter

@Configuration(proxyBeanMethods = false)
class MyWebSocketConfiguration {

    @Bean
    fun serverEndpointExporter(): ServerEndpointExporter {
        return ServerEndpointExporter()
    }

}
```

The bean shown in the preceding example registers any `@ServerEndpoint` annotated beans with the underlying WebSocket container. When deployed to a standalone servlet container, this role is performed by a servlet container initializer, and the `ServerEndpointExporter` bean is not required.

18.4. Spring MVC

Spring Boot has a number of starters that include Spring MVC. Note that some starters include a dependency on Spring MVC rather than include it directly. This section answers common questions about Spring MVC and Spring Boot.

18.4.1. Write a JSON REST Service

Any Spring `@RestController` in a Spring Boot application should render JSON response by default as long as Jackson2 is on the classpath, as shown in the following example:

Java

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MyController {

    @RequestMapping("/thing")
    public MyThing thing() {
        return new MyThing();
    }

}
```

Kotlin

```
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@RestController
class MyController {

    @RequestMapping("/thing")
    fun thing(): MyThing {
        return MyThing()
    }

}
```

As long as `MyThing` can be serialized by Jackson2 (true for a normal POJO or Groovy object), then `localhost:8080/thing` serves a JSON representation of it by default. Note that, in a browser, you might sometimes see XML responses, because browsers tend to send accept headers that prefer XML.

18.4.2. Write an XML REST Service

If you have the Jackson XML extension (`jackson-dataformat-xml`) on the classpath, you can use it to render XML responses. The previous example that we used for JSON would work. To use the Jackson XML renderer, add the following dependency to your project:

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

If Jackson's XML extension is not available and JAXB is available, XML can be rendered with the additional requirement of having `MyThing` annotated as `@XmlRootElement`, as shown in the following example:

Java

```
import jakarta.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class MyThing {

    private String name;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

Kotlin

```
import jakarta.xml.bind.annotation.XmlRootElement

@XmlRootElement
class MyThing {

    var name: String? = null

}
```

You will need to ensure that the JAXB library is part of your project, for example by adding:

```
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
</dependency>
```

NOTE To get the server to render XML instead of JSON, you might have to send an `Accept: text/xml` header (or use a browser).

18.4.3. Customize the Jackson ObjectMapper

Spring MVC (client and server side) uses `HttpMessageConverters` to negotiate content conversion in an HTTP exchange. If Jackson is on the classpath, you already get the default converter(s) provided by `Jackson2ObjectMapperBuilder`, an instance of which is auto-configured for you.

The `ObjectMapper` (or `XmlMapper` for Jackson XML converter) instance (created by default) has the following customized properties:

- `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled
- `SerializationFeature.WRITE_DATES_AS_TIMESTAMPS` is disabled
- `SerializationFeature.WRITE_DURATIONS_AS_TIMESTAMPS` is disabled

Spring Boot also has some features to make it easier to customize this behavior.

You can configure the `ObjectMapper` and `XmlMapper` instances by using the environment. Jackson provides an extensive suite of on/off features that can be used to configure various aspects of its processing. These features are described in six enums (in Jackson) that map onto properties in the environment:

Enum	Property	Values
`com.fasterxml.jackson.databind.DeserializationFeature`	`spring.jackson.deserialization.<feature_name>`	`true`, `false`
`com.fasterxml.jackson.core.JsonGenerator.Feature`	`spring.jackson.generator.<feature_name>`	`true`, `false`
`com.fasterxml.jackson.databind.MapperFeature`	`spring.jackson.mapper.<feature_name>`	`true`, `false`
`com.fasterxml.jackson.core.JsonParser.Feature`	`spring.jackson.parser.<feature_name>`	`true`, `false`
`com.fasterxml.jackson.databind.SerializationFeature`	`spring.jackson.serialization.<feature_name>`	`true`, `false`
`com.fasterxml.jackson.annotation.JsonInclude.Include`	`spring.jackson.default-property-inclusion`	`always`, `non_null`, `non_absent`, `non_default`, `non_empty`

For example, to enable pretty print, set `spring.jackson.serialization.indent_output=true`. Note that, thanks to the use of relaxed binding, the case of `indent_output` does not have to match the case of the corresponding enum constant, which is `INDENT_OUTPUT`.

This environment-based configuration is applied to the auto-configured `Jackson2ObjectMapperBuilder` bean and applies to any mappers created by using the builder, including the auto-configured `ObjectMapper` bean.

The context's `Jackson2ObjectMapperBuilder` can be customized by one or more `Jackson2ObjectMapperBuilderCustomizer` beans. Such customizer beans can be ordered (Boot's own customizer has an order of 0), letting additional customization be applied both before and after Boot's customization.

Any beans of type `com.fasterxml.jackson.databind.Module` are automatically registered with the auto-configured `Jackson2ObjectMapperBuilder` and are applied to any `ObjectMapper` instances that it creates. This provides a global mechanism for contributing custom modules when you add new features to your application.

If you want to replace the default `ObjectMapper` completely, either define a `@Bean` of that type and mark it as `@Primary` or, if you prefer the builder-based approach, define a `Jackson2ObjectMapperBuilder` `@Bean`. Note that, in either case, doing so disables all auto-configuration

of the `ObjectMapper`.

If you provide any `@Beans` of type `MappingJackson2HttpMessageConverter`, they replace the default value in the MVC configuration. Also, a convenience bean of type `HttpMessageConverters` is provided (and is always available if you use the default MVC configuration). It has some useful methods to access the default and user-enhanced message converters.

See the "Customize the @ResponseBody Rendering" section and the `WebMvcAutoConfiguration` source code for more details.

18.4.4. Customize the @ResponseBody Rendering

Spring uses `HttpMessageConverters` to render `@ResponseBody` (or responses from `@RestController`). You can contribute additional converters by adding beans of the appropriate type in a Spring Boot context. If a bean you add is of a type that would have been included by default anyway (such as `MappingJackson2HttpMessageConverter` for JSON conversions), it replaces the default value. A convenience bean of type `HttpMessageConverters` is provided and is always available if you use the default MVC configuration. It has some useful methods to access the default and user-enhanced message converters (For example, it can be useful if you want to manually inject them into a custom `RestTemplate`).

As in normal MVC usage, any `WebMvcConfigurer` beans that you provide can also contribute converters by overriding the `configureMessageConverters` method. However, unlike with normal MVC, you can supply only additional converters that you need (because Spring Boot uses the same mechanism to contribute its defaults). Finally, if you opt out of the Spring Boot default MVC configuration by providing your own `@EnableWebMvc` configuration, you can take control completely and do everything manually by using `getMessageConverters` from `WebMvcConfigurationSupport`.

See the `WebMvcAutoConfiguration` source code for more details.

18.4.5. Handling Multipart File Uploads

Spring Boot embraces the servlet 5 `jakarta.servlet.http.Part` API to support uploading files. By default, Spring Boot configures Spring MVC with a maximum size of 1MB per file and a maximum of 10MB of file data in a single request. You may override these values, the location to which intermediate data is stored (for example, to the `/tmp` directory), and the threshold past which data is flushed to disk by using the properties exposed in the `MultipartProperties` class. For example, if you want to specify that files be unlimited, set the `spring.servlet.multipart.max-file-size` property to `-1`.

The multipart support is helpful when you want to receive multipart encoded file data as a `@RequestParam`-annotated parameter of type `MultipartFile` in a Spring MVC controller handler method.

See the `MultipartAutoConfiguration` source for more details.

NOTE	It is recommended to use the container's built-in support for multipart uploads rather than introducing an additional dependency such as Apache Commons File Upload.

18.4.6. Switch Off the Spring MVC DispatcherServlet

By default, all content is served from the root of your application (/). If you would rather map to a different path, you can configure one as follows:

Properties

```
spring.mvc.servlet.path=/mypath
```

Yaml

```
spring:
  mvc:
    servlet:
      path: "/mypath"
```

If you have additional servlets you can declare a `@Bean` of type `Servlet` or `ServletRegistrationBean` for each and Spring Boot will register them transparently to the container. Because servlets are registered that way, they can be mapped to a sub-context of the `DispatcherServlet` without invoking it.

Configuring the `DispatcherServlet` yourself is unusual but if you really need to do it, a `@Bean` of type `DispatcherServletPath` must be provided as well to provide the path of your custom `DispatcherServlet`.

18.4.7. Switch off the Default MVC Configuration

The easiest way to take complete control over MVC configuration is to provide your own `@Configuration` with the `@EnableWebMvc` annotation. Doing so leaves all MVC configuration in your hands.

18.4.8. Customize ViewResolvers

A `ViewResolver` is a core component of Spring MVC, translating view names in `@Controller` to actual `View` implementations. Note that `ViewResolvers` are mainly used in UI applications, rather than REST-style services (a `View` is not used to render a `@ResponseBody`). There are many implementations of `ViewResolver` to choose from, and Spring on its own is not opinionated about which ones you should use. Spring Boot, on the other hand, installs one or two for you, depending on what it finds on the classpath and in the application context. The `DispatcherServlet` uses all the resolvers it finds in the application context, trying each one in turn until it gets a result. If you add your own, you have to be aware of the order and in which position your resolver is added.

`WebMvcAutoConfiguration` adds the following `ViewResolvers` to your context:

- An `InternalResourceViewResolver` named 'defaultViewResolver'. This one locates physical resources that can be rendered by using the `DefaultServlet` (including static resources and JSP pages, if you use those). It applies a prefix and a suffix to the view name and then looks for a physical resource with that path in the servlet context (the defaults are both empty but are accessible for external configuration through `spring.mvc.view.prefix` and

`spring.mvc.view.suffix`). You can override it by providing a bean of the same type.

- A `BeanNameViewResolver` named 'beanNameViewResolver'. This is a useful member of the view resolver chain and picks up any beans with the same name as the `View` being resolved. It should not be necessary to override or replace it.
- A `ContentNegotiatingViewResolver` named 'viewResolver' is added only if there **are** actually beans of type `View` present. This is a composite resolver, delegating to all the others and attempting to find a match to the 'Accept' HTTP header sent by the client. There is a useful blog about `ContentNegotiatingViewResolver` that you might like to study to learn more, and you might also look at the source code for detail. You can switch off the auto-configured `ContentNegotiatingViewResolver` by defining a bean named 'viewResolver'.
- If you use Thymeleaf, you also have a `ThymeleafViewResolver` named 'thymeleafViewResolver'. It looks for resources by surrounding the view name with a prefix and suffix. The prefix is `spring.thymeleaf.prefix`, and the suffix is `spring.thymeleaf.suffix`. The values of the prefix and suffix default to 'classpath:/templates/' and '.html', respectively. You can override `ThymeleafViewResolver` by providing a bean of the same name.
- If you use FreeMarker, you also have a `FreeMarkerViewResolver` named 'freeMarkerViewResolver'. It looks for resources in a loader path (which is externalized to `spring.freemarker.templateLoaderPath` and has a default value of 'classpath:/templates/') by surrounding the view name with a prefix and a suffix. The prefix is externalized to `spring.freemarker.prefix`, and the suffix is externalized to `spring.freemarker.suffix`. The default values of the prefix and suffix are empty and '.ftlh', respectively. You can override `FreeMarkerViewResolver` by providing a bean of the same name.
- If you use Groovy templates (actually, if `groovy-templates` is on your classpath), you also have a `GroovyMarkupViewResolver` named 'groovyMarkupViewResolver'. It looks for resources in a loader path by surrounding the view name with a prefix and suffix (externalized to `spring.groovy.template.prefix` and `spring.groovy.template.suffix`). The prefix and suffix have default values of 'classpath:/templates/' and '.tpl', respectively. You can override `GroovyMarkupViewResolver` by providing a bean of the same name.
- If you use Mustache, you also have a `MustacheViewResolver` named 'mustacheViewResolver'. It looks for resources by surrounding the view name with a prefix and suffix. The prefix is `spring.mustache.prefix`, and the suffix is `spring.mustache.suffix`. The values of the prefix and suffix default to 'classpath:/templates/' and '.mustache', respectively. You can override `MustacheViewResolver` by providing a bean of the same name.

For more detail, see the following sections:

- `WebMvcAutoConfiguration`
- `ThymeleafAutoConfiguration`
- `FreeMarkerAutoConfiguration`
- `GroovyTemplateAutoConfiguration`

18.5. Jersey

18.5.1. Secure Jersey endpoints with Spring Security

Spring Security can be used to secure a Jersey-based web application in much the same way as it can be used to secure a Spring MVC-based web application. However, if you want to use Spring Security's method-level security with Jersey, you must configure Jersey to use `setStatus(int)` rather `sendError(int)`. This prevents Jersey from committing the response before Spring Security has had an opportunity to report an authentication or authorization failure to the client.

The `jersey.config.server.response.setStatusOverSendError` property must be set to `true` on the application's `ResourceConfig` bean, as shown in the following example:

```
import java.util.Collections;

import org.glassfish.jersey.server.ResourceConfig;

import org.springframework.stereotype.Component;

@Component
public class JerseySetStatusOverSendErrorConfig extends ResourceConfig {

    public JerseySetStatusOverSendErrorConfig() {
        register(Endpoint.class);

setProperties(Collections.singletonMap("jersey.config.server.response.setStatusOverSen
dError", true));
    }

}
```

18.5.2. Use Jersey Alongside Another Web Framework

To use Jersey alongside another web framework, such as Spring MVC, it should be configured so that it will allow the other framework to handle requests that it cannot handle. First, configure Jersey to use a filter rather than a servlet by configuring the `spring.jersey.type` application property with a value of `filter`. Second, configure your `ResourceConfig` to forward requests that would have resulted in a 404, as shown in the following example.

```
import org.glassfish.jersey.server.ResourceConfig;
import org.glassfish.jersey.servlet.ServletProperties;

import org.springframework.stereotype.Component;

@Component
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        register(Endpoint.class);
        property(ServletProperties.FILTER_FORWARD_ON_404, true);
    }

}
```

18.6. HTTP Clients

Spring Boot offers a number of starters that work with HTTP clients. This section answers questions related to using them.

18.6.1. Configure RestTemplate to Use a Proxy

As described in RestTemplate Customization, you can use a `RestTemplateCustomizer` with `RestTemplateBuilder` to build a customized `RestTemplate`. This is the recommended approach for creating a `RestTemplate` configured to use a proxy.

The exact details of the proxy configuration depend on the underlying client request factory that is being used.

18.6.2. Configure the TcpClient used by a Reactor Netty-based WebClient

When Reactor Netty is on the classpath a Reactor Netty-based `WebClient` is auto-configured. To customize the client's handling of network connections, provide a `ClientHttpConnector` bean. The following example configures a 60 second connect timeout and adds a `ReadTimeoutHandler`:

Java

```
import io.netty.channel.ChannelOption;
import io.netty.handler.timeout.ReadTimeoutHandler;
import reactor.netty.http.client.HttpClient;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.client.reactive.ClientHttpConnector;
import org.springframework.http.client.reactive.ReactorClientHttpConnector;
import org.springframework.http.client.reactive.ReactorResourceFactory;

@Configuration(proxyBeanMethods = false)
public class MyReactorNettyClientConfiguration {

    @Bean
    ClientHttpConnector clientHttpConnector(ReactorResourceFactory resourceFactory) {
        HttpClient httpClient =
HttpClient.create(resourceFactory.getConnectionProvider())
                .runOn(resourceFactory.getLoopResources())
                .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 60000)
                .doOnConnected((connection) -> connection.addHandlerLast(new
ReadTimeoutHandler(60)));
        return new ReactorClientHttpConnector(httpClient);
    }

}
```

Kotlin

```
import io.netty.channel.ChannelOption
import io.netty.handler.timeout.ReadTimeoutHandler
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.client.reactive.ClientHttpConnector
import org.springframework.http.client.reactive.ReactorClientHttpConnector
import org.springframework.http.client.reactive.ReactorResourceFactory
import reactor.netty.http.client.HttpClient

@Configuration(proxyBeanMethods = false)
class MyReactorNettyClientConfiguration {

    @Bean
    fun clientHttpConnector(resourceFactory: ReactorResourceFactory):
ClientHttpConnector {
        val httpClient = HttpClient.create(resourceFactory.connectionProvider)
            .runOn(resourceFactory.loopResources)
            .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 60000)
            .doOnConnected { connection ->
                connection.addHandlerLast(ReadTimeoutHandler(60))
            }
        return ReactorClientHttpConnector(httpClient)
    }

}
```

TIP Note the use of `ReactorResourceFactory` for the connection provider and event loop resources. This ensures efficient sharing of resources for the server receiving requests and the client making requests.

18.7. Logging

Spring Boot has no mandatory logging dependency, except for the Commons Logging API, which is typically provided by Spring Framework's `spring-jcl` module. To use Logback, you need to include it and `spring-jcl` on the classpath. The recommended way to do that is through the starters, which all depend on `spring-boot-starter-logging`. For a web application, you need only `spring-boot-starter-web`, since it depends transitively on the logging starter. If you use Maven, the following dependency adds logging for you:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Boot has a `LoggingSystem` abstraction that attempts to configure logging based on the content

of the classpath. If Logback is available, it is the first choice.

If the only change you need to make to logging is to set the levels of various loggers, you can do so in `application.properties` by using the "logging.level" prefix, as shown in the following example:

Properties

```
logging.level.org.springframework.web=debug
logging.level.org.hibernate=error
```

Yaml

```
logging:
  level:
    org.springframework.web: "debug"
    org.hibernate: "error"
```

You can also set the location of a file to which to write the log (in addition to the console) by using `logging.file.name`.

To configure the more fine-grained settings of a logging system, you need to use the native configuration format supported by the `LoggingSystem` in question. By default, Spring Boot picks up the native configuration from its default location for the system (such as `classpath:logback.xml` for Logback), but you can set the location of the config file by using the `logging.config` property.

18.7.1. Configure Logback for Logging

If you need to apply customizations to logback beyond those that can be achieved with `application.properties`, you will need to add a standard logback configuration file. You can add a `logback.xml` file to the root of your classpath for logback to find. You can also use `logback-spring.xml` if you want to use the Spring Boot Logback extensions.

TIP The Logback documentation has a dedicated section that covers configuration in some detail.

Spring Boot provides a number of logback configurations that can be `included` in your own configuration. These includes are designed to allow certain common Spring Boot conventions to be re-applied.

The following files are provided under `org/springframework/boot/logging/logback/`:

- `defaults.xml` - Provides conversion rules, pattern properties and common logger configurations.
- `console-appender.xml` - Adds a `ConsoleAppender` using the `CONSOLE_LOG_PATTERN`.
- `file-appender.xml` - Adds a `RollingFileAppender` using the `FILE_LOG_PATTERN` and `ROLLING_FILE_NAME_PATTERN` with appropriate settings.

In addition, a legacy `base.xml` file is provided for compatibility with earlier versions of Spring Boot.

A typical custom `logback.xml` file would look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
    <include resource="org/springframework/boot/logging/logback/console-appender.xml"
/>
    <root level="INFO">
        <appender-ref ref="CONSOLE" />
    </root>
    <logger name="org.springframework.web" level="DEBUG"/>
</configuration>
```

Your logback configuration file can also make use of System properties that the `LoggingSystem` takes care of creating for you:

- `${PID}`: The current process ID.
- `${LOG_FILE}`: Whether `logging.file.name` was set in Boot's external configuration.
- `${LOG_PATH}`: Whether `logging.file.path` (representing a directory for log files to live in) was set in Boot's external configuration.
- `${LOG_EXCEPTION_CONVERSION_WORD}`: Whether `logging.exception-conversion-word` was set in Boot's external configuration.
- `${ROLLING_FILE_NAME_PATTERN}`: Whether `logging.pattern.rolling-file-name` was set in Boot's external configuration.

Spring Boot also provides some nice ANSI color terminal output on a console (but not in a log file) by using a custom Logback converter. See the `CONSOLE_LOG_PATTERN` in the `defaults.xml` configuration for an example.

If Groovy is on the classpath, you should be able to configure Logback with `logback.groovy` as well. If present, this setting is given preference.

NOTE	Spring extensions are not supported with Groovy configuration. Any `logback-spring.groovy` files will not be detected.

Configure Logback for File-only Output

If you want to disable console logging and write output only to a file, you need a custom `logback-spring.xml` that imports `file-appender.xml` but not `console-appender.xml`, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml" />
    <property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-
${java.io.tmpdir:-/tmp}}/}spring.log}"/>
    <include resource="org/springframework/boot/logging/logback/file-appender.xml" />
    <root level="INFO">
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

You also need to add `logging.file.name` to your `application.properties` or `application.yaml`, as shown in the following example:

Properties

```
logging.file.name=myapplication.log
```

Yaml

```
logging:
  file:
    name: "myapplication.log"
```

18.7.2. Configure Log4j for Logging

Spring Boot supports Log4j 2 for logging configuration if it is on the classpath. If you use the starters for assembling dependencies, you have to exclude Logback and then include Log4j 2 instead. If you do not use the starters, you need to provide (at least) `spring-jcl` in addition to Log4j 2.

The recommended path is through the starters, even though it requires some jiggling. The following example shows how to set up the starters in Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

Gradle provides a few different ways to set up the starters. One way is to use a module replacement. To do so, declare a dependency on the Log4j 2 starter and tell Gradle that any occurrences of the default logging starter should be replaced by the Log4j 2 starter, as shown in the following example:

```
dependencies {
    implementation "org.springframework.boot:spring-boot-starter-log4j2"
    modules {
        module("org.springframework.boot:spring-boot-starter-logging") {
            replacedBy("org.springframework.boot:spring-boot-starter-log4j2", "Use
Log4j2 instead of Logback")
        }
    }
}
```

NOTE	The Log4j starters gather together the dependencies for common logging requirements (such as having Tomcat use `java.util.logging` but configuring the output using Log4j 2).
NOTE	To ensure that debug logging performed using `java.util.logging` is routed into Log4j 2, configure its JDK logging adapter by setting the `java.util.logging.manager` system property to `org.apache.logging.log4j.jul.LogManager`.

Use YAML or JSON to Configure Log4j 2

In addition to its default XML configuration format, Log4j 2 also supports YAML and JSON configuration files. To configure Log4j 2 to use an alternative configuration file format, add the appropriate dependencies to the classpath and name your configuration files to match your chosen

file format, as shown in the following example:

Format	Dependencies	File names
YAML	`com.fasterxml.jackson.core:jackson-databind` + `com.fasterxml.jackson.dataformat:jackson-dataformat-yaml`	`log4j2.yaml` + `log4j2.yml`
JSON	`com.fasterxml.jackson.core:jackson-databind`	`log4j2.json` + `log4j2.jsn`

Use Composite Configuration to Configure Log4j 2

Log4j 2 has support for combining multiple configuration files into a single composite configuration. To use this support in Spring Boot, configure `logging.log4j2.config.override` with the locations of one or more secondary configuration files. The secondary configuration files will be merged with the primary configuration, whether the primary's source is Spring Boot's defaults, a standard location such as `log4j.xml`, or the location configured by the `logging.config` property.

18.8. Data Access

Spring Boot includes a number of starters for working with data sources. This section answers questions related to doing so.

18.8.1. Configure a Custom DataSource

To configure your own `DataSource`, define a `@Bean` of that type in your configuration. Spring Boot reuses your `DataSource` anywhere one is required, including database initialization. If you need to externalize some settings, you can bind your `DataSource` to the environment (see "Third-party Configuration").

The following example shows how to define a data source in a bean:

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyDataSourceConfiguration {

    @Bean
    @ConfigurationProperties(prefix = "app.datasource")
    public SomeDataSource dataSource() {
        return new SomeDataSource();
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyDataSourceConfiguration {

    @Bean
    @ConfigurationProperties(prefix = "app.datasource")
    fun dataSource(): SomeDataSource {
        return SomeDataSource()
    }

}
```

The following example shows how to define a data source by setting properties:

Properties

```
app.datasource.url=jdbc:h2:mem:mydb
app.datasource.username=sa
app.datasource.pool-size=30
```

Yaml

```
app:
  datasource:
    url: "jdbc:h2:mem:mydb"
    username: "sa"
    pool-size: 30
```

Assuming that `SomeDataSource` has regular JavaBean properties for the URL, the username, and the pool size, these settings are bound automatically before the `DataSource` is made available to other components.

Spring Boot also provides a utility builder class, called `DataSourceBuilder`, that can be used to create one of the standard data sources (if it is on the classpath). The builder can detect the one to use based on what is available on the classpath. It also auto-detects the driver based on the JDBC URL.

The following example shows how to create a data source by using a `DataSourceBuilder`:

Java

```
import javax.sql.DataSource;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyDataSourceConfiguration {

    @Bean
    @ConfigurationProperties("app.datasource")
    public DataSource dataSource() {
        return DataSourceBuilder.create().build();
    }

}
```

Kotlin

```
import javax.sql.DataSource

import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.jdbc.DataSourceBuilder
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyDataSourceConfiguration {

    @Bean
    @ConfigurationProperties("app.datasource")
    fun dataSource(): DataSource {
        return DataSourceBuilder.create().build()
    }

}
```

To run an app with that `DataSource`, all you need is the connection information. Pool-specific settings can also be provided. Check the implementation that is going to be used at runtime for more details.

The following example shows how to define a JDBC data source by setting properties:

Properties

```
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.pool-size=30
```

Yaml

```
app:
  datasource:
    url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    password: "dbpass"
    pool-size: 30
```

However, there is a catch. Because the actual type of the connection pool is not exposed, no keys are generated in the metadata for your custom `DataSource` and no completion is available in your IDE (because the `DataSource` interface exposes no properties). Also, if you happen to have Hikari on the classpath, this basic setup does not work, because Hikari has no `url` property (but does have a `jdbcUrl` property). In that case, you must rewrite your configuration as follows:

Properties

```
app.datasource.jdbc-url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.pool-size=30
```

Yaml

```
app:
  datasource:
    jdbc-url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    password: "dbpass"
    pool-size: 30
```

You can fix that by forcing the connection pool to use and return a dedicated implementation rather than `DataSource`. You cannot change the implementation at runtime, but the list of options will be explicit.

The following example shows how create a `HikariDataSource` with `DataSourceBuilder`:

Java

```
import com.zaxxer.hikari.HikariDataSource;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyDataSourceConfiguration {

    @Bean
    @ConfigurationProperties("app.datasource")
    public HikariDataSource dataSource() {
        return DataSourceBuilder.create().type(HikariDataSource.class).build();
    }

}
```

Kotlin

```
import com.zaxxer.hikari.HikariDataSource
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.jdbc.DataSourceBuilder
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyDataSourceConfiguration {

    @Bean
    @ConfigurationProperties("app.datasource")
    fun dataSource(): HikariDataSource {
        return DataSourceBuilder.create().type(HikariDataSource::class.java).build()
    }

}
```

You can even go further by leveraging what `DataSourceProperties` does for you — that is, by providing a default embedded database with a sensible username and password if no URL is provided. You can easily initialize a `DataSourceBuilder` from the state of any `DataSourceProperties` object, so you could also inject the DataSource that Spring Boot creates automatically. However, that would split your configuration into two namespaces: `url`, `username`, `password`, `type`, and `driver` on `spring.datasource` and the rest on your custom namespace (`app.datasource`). To avoid that, you can redefine a custom `DataSourceProperties` on your custom namespace, as shown in the following example:

Java

```
import com.zaxxer.hikari.HikariDataSource;

import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration(proxyBeanMethods = false)
public class MyDataSourceConfiguration {

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource")
    public DataSourceProperties dataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    @ConfigurationProperties("app.datasource.configuration")
    public HikariDataSource dataSource(DataSourceProperties properties) {
        return
properties.initializeDataSourceBuilder().type(HikariDataSource.class).build();
    }

}
```

Kotlin

```
import com.zaxxer.hikari.HikariDataSource
import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Primary

@Configuration(proxyBeanMethods = false)
class MyDataSourceConfiguration {

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource")
    fun dataSourceProperties(): DataSourceProperties {
        return DataSourceProperties()
    }

    @Bean
    @ConfigurationProperties("app.datasource.configuration")
    fun dataSource(properties: DataSourceProperties): HikariDataSource {
        return
properties.initializeDataSourceBuilder().type(HikariDataSource::class.java).build()
    }

}
```

This setup puts you *in sync* with what Spring Boot does for you by default, except that a dedicated connection pool is chosen (in code) and its settings are exposed in the `app.datasource.configuration` sub namespace. Because `DataSourceProperties` is taking care of the `url`/`jdbcUrl` translation for you, you can configure it as follows:

Properties

```
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.configuration.maximum-pool-size=30
```

Yaml

```
app:
  datasource:
    url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    password: "dbpass"
    configuration:
      maximum-pool-size: 30
```

TIP	Spring Boot will expose Hikari-specific settings to `spring.datasource.hikari`. This example uses a more generic `configuration` sub namespace as the example does not support multiple datasource implementations.

NOTE	Because your custom configuration chooses to go with Hikari, `app.datasource.type` has no effect. In practice, the builder is initialized with whatever value you might set there and then overridden by the call to `.type()`.

See "Configure a DataSource" in the "Spring Boot features" section and the `DataSourceAutoConfiguration` class for more details.

18.8.2. Configure Two DataSources

If you need to configure multiple data sources, you can apply the same tricks that are described in the previous section. You must, however, mark one of the `DataSource` instances as `@Primary`, because various auto-configurations down the road expect to be able to get one by type.

If you create your own `DataSource`, the auto-configuration backs off. In the following example, we provide the *exact* same feature set as the auto-configuration provides on the primary data source:

Java

```
import com.zaxxer.hikari.HikariDataSource;
import org.apache.commons.dbcp2.BasicDataSource;

import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration(proxyBeanMethods = false)
public class MyDataSourcesConfiguration {

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource.first")
    public DataSourceProperties firstDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource.first.configuration")
    public HikariDataSource firstDataSource(DataSourceProperties
firstDataSourceProperties) {
        return
firstDataSourceProperties.initializeDataSourceBuilder().type(HikariDataSource.class).b
uild();
    }

    @Bean
    @ConfigurationProperties("app.datasource.second")
    public BasicDataSource secondDataSource() {
        return DataSourceBuilder.create().type(BasicDataSource.class).build();
    }

}
```

Kotlin

```
import com.zaxxer.hikari.HikariDataSource
import org.apache.commons.dbcp2.BasicDataSource
import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.jdbc.DataSourceBuilder
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Primary

@Configuration(proxyBeanMethods = false)
class MyDataSourcesConfiguration {

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource.first")
    fun firstDataSourceProperties(): DataSourceProperties {
        return DataSourceProperties()
    }

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource.first.configuration")
    fun firstDataSource(firstDataSourceProperties: DataSourceProperties):
HikariDataSource {
        return
firstDataSourceProperties.initializeDataSourceBuilder().type(HikariDataSource::class.j
ava).build()
    }

    @Bean
    @ConfigurationProperties("app.datasource.second")
    fun secondDataSource(): BasicDataSource {
        return DataSourceBuilder.create().type(BasicDataSource::class.java).build()
    }

}
```

TIP `firstDataSourceProperties` has to be flagged as `@Primary` so that the database initializer feature uses your copy (if you use the initializer).

Both data sources are also bound for advanced customizations. For instance, you could configure them as follows:

Properties

```
app.datasource.first.url=jdbc:mysql://localhost/first
app.datasource.first.username=dbuser
app.datasource.first.password=dbpass
app.datasource.first.configuration.maximum-pool-size=30

app.datasource.second.url=jdbc:mysql://localhost/second
app.datasource.second.username=dbuser
app.datasource.second.password=dbpass
app.datasource.second.max-total=30
```

Yaml

```
app:
  datasource:
    first:
      url: "jdbc:mysql://localhost/first"
      username: "dbuser"
      password: "dbpass"
      configuration:
        maximum-pool-size: 30

    second:
      url: "jdbc:mysql://localhost/second"
      username: "dbuser"
      password: "dbpass"
      max-total: 30
```

You can apply the same concept to the secondary `DataSource` as well, as shown in the following example:

Java

```
import com.zaxxer.hikari.HikariDataSource;
import org.apache.commons.dbcp2.BasicDataSource;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration(proxyBeanMethods = false)
public class MyCompleteDataSourcesConfiguration {

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource.first")
    public DataSourceProperties firstDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource.first.configuration")
    public HikariDataSource firstDataSource(DataSourceProperties
firstDataSourceProperties) {
        return
firstDataSourceProperties.initializeDataSourceBuilder().type(HikariDataSource.class).b
uild();
    }

    @Bean
    @ConfigurationProperties("app.datasource.second")
    public DataSourceProperties secondDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean
    @ConfigurationProperties("app.datasource.second.configuration")
    public BasicDataSource secondDataSource(
            @Qualifier("secondDataSourceProperties") DataSourceProperties
secondDataSourceProperties) {
        return
secondDataSourceProperties.initializeDataSourceBuilder().type(BasicDataSource.class).b
uild();
    }

}
```

Kotlin

```
import com.zaxxer.hikari.HikariDataSource
import org.apache.commons.dbcp2.BasicDataSource
import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Primary

@Configuration(proxyBeanMethods = false)
class MyCompleteDataSourcesConfiguration {

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource.first")
    fun firstDataSourceProperties(): DataSourceProperties {
        return DataSourceProperties()
    }

    @Bean
    @Primary
    @ConfigurationProperties("app.datasource.first.configuration")
    fun firstDataSource(firstDataSourceProperties: DataSourceProperties):
HikariDataSource {
        return
firstDataSourceProperties.initializeDataSourceBuilder().type(HikariDataSource::class.j
ava).build()
    }

    @Bean
    @ConfigurationProperties("app.datasource.second")
    fun secondDataSourceProperties(): DataSourceProperties {
        return DataSourceProperties()
    }

    @Bean
    @ConfigurationProperties("app.datasource.second.configuration")
    fun secondDataSource(secondDataSourceProperties: DataSourceProperties):
BasicDataSource {
        return
secondDataSourceProperties.initializeDataSourceBuilder().type(BasicDataSource::class.j
ava).build()
    }

}
```

The preceding example configures two data sources on custom namespaces with the same logic as Spring Boot would use in auto-configuration. Note that each `configuration` sub namespace provides advanced settings based on the chosen implementation.

18.8.3. Use Spring Data Repositories

Spring Data can create implementations of `@Repository` interfaces of various flavors. Spring Boot handles all of that for you, as long as those `@Repositories` are included in one of the auto-configuration packages, typically the package (or a sub-package) of your main application class that is annotated with `@SpringBootApplication` or `@EnableAutoConfiguration`.

For many applications, all you need is to put the right Spring Data dependencies on your classpath. There is a `spring-boot-starter-data-jpa` for JPA, `spring-boot-starter-data-mongodb` for Mongodb, and various other starters for supported technologies. To get started, create some repository interfaces to handle your `@Entity` objects.

Spring Boot determines the location of your `@Repository` definitions by scanning the auto-configuration packages. For more control, use the `@Enable···Repositories` annotations from Spring Data.

For more about Spring Data, see the Spring Data project page.

18.8.4. Separate @Entity Definitions from Spring Configuration

Spring Boot determines the location of your `@Entity` definitions by scanning the auto-configuration packages. For more control, use the `@EntityScan` annotation, as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
@EnableAutoConfiguration
@EntityScan(basePackageClasses = City.class)
public class MyApplication {

    // ...

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.EnableAutoConfiguration
import org.springframework.boot.autoconfigure.domain.EntityScan
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
@EnableAutoConfiguration
@EntityScan(basePackageClasses = [City::class])
class MyApplication {

    // ...

}
```

18.8.5. Configure JPA Properties

Spring Data JPA already provides some vendor-independent configuration options (such as those for SQL logging), and Spring Boot exposes those options and a few more for Hibernate as external configuration properties. Some of them are automatically detected according to the context so you should not have to set them.

The `spring.jpa.hibernate.ddl-auto` is a special case, because, depending on runtime conditions, it has different defaults. If an embedded database is used and no schema manager (such as Liquibase or Flyway) is handling the `DataSource`, it defaults to `create-drop`. In all other cases, it defaults to `none`.

The dialect to use is detected by the JPA provider. If you prefer to set the dialect yourself, set the `spring.jpa.database-platform` property.

The most common options to set are shown in the following example:

Properties

```
spring.jpa.hibernate.naming.physical-strategy=com.example.MyPhysicalNamingStrategy
spring.jpa.show-sql=true
```

Yaml

```
spring:
  jpa:
    hibernate:
      naming:
        physical-strategy: "com.example.MyPhysicalNamingStrategy"
    show-sql: true
```

In addition, all properties in `spring.jpa.properties.*` are passed through as normal JPA properties (with the prefix stripped) when the local `EntityManagerFactory` is created.

WARNING	You need to ensure that names defined under `spring.jpa.properties.*` exactly match those expected by your JPA provider. Spring Boot will not attempt any kind of relaxed binding for these entries. For example, if you want to configure Hibernate's batch size you must use `spring.jpa.properties.hibernate.jdbc.batch_size`. If you use other forms, such as `batchSize` or `batch-size`, Hibernate will not apply the setting.

TIP	If you need to apply advanced customization to Hibernate properties, consider registering a `HibernatePropertiesCustomizer` bean that will be invoked prior to creating the `EntityManagerFactory`. This takes precedence to anything that is applied by the auto-configuration.

18.8.6. Configure Hibernate Naming Strategy

Hibernate uses two different naming strategies to map names from the object model to the corresponding database names. The fully qualified class name of the physical and the implicit strategy implementations can be configured by setting the `spring.jpa.hibernate.naming.physical-strategy` and `spring.jpa.hibernate.naming.implicit-strategy` properties, respectively. Alternatively, if `ImplicitNamingStrategy` or `PhysicalNamingStrategy` beans are available in the application context, Hibernate will be automatically configured to use them.

By default, Spring Boot configures the physical naming strategy with `CamelCaseToUnderscoresNamingStrategy`. Using this strategy, all dots are replaced by underscores and camel casing is replaced by underscores as well. Additionally, by default, all table names are generated in lower case. For example, a `TelephoneNumber` entity is mapped to the `telephone_number` table. If your schema requires mixed-case identifiers, define a custom `CamelCaseToUnderscoresNamingStrategy` bean, as shown in the following example:

Java

```
import org.hibernate.boot.model.naming.CamelCaseToUnderscoresNamingStrategy;
import org.hibernate.engine.jdbc.env.spi.JdbcEnvironment;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyHibernateConfiguration {

    @Bean
    public CamelCaseToUnderscoresNamingStrategy caseSensitivePhysicalNamingStrategy()
{
        return new CamelCaseToUnderscoresNamingStrategy() {

            @Override
            protected boolean isCaseInsensitive(JdbcEnvironment jdbcEnvironment) {
                return false;
            }

        };
    }

}
```

Kotlin

```
import org.hibernate.boot.model.naming.CamelCaseToUnderscoresNamingStrategy
import org.hibernate.engine.jdbc.env.spi.JdbcEnvironment
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyHibernateConfiguration {

    @Bean
    fun caseSensitivePhysicalNamingStrategy(): CamelCaseToUnderscoresNamingStrategy {
        return object : CamelCaseToUnderscoresNamingStrategy() {
            override fun isCaseInsensitive(jdbcEnvironment: JdbcEnvironment): Boolean
{
                return false
            }
        }
    }

}
```

If you prefer to use Hibernate's default instead, set the following property:

```
spring.jpa.hibernate.naming.physical-
strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

Alternatively, you can configure the following bean:

Java

```
import org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
class MyHibernateConfiguration {

    @Bean
    PhysicalNamingStrategyStandardImpl caseSensitivePhysicalNamingStrategy() {
        return new PhysicalNamingStrategyStandardImpl();
    }

}
```

Kotlin

```
import org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
internal class MyHibernateConfiguration {

    @Bean
    fun caseSensitivePhysicalNamingStrategy(): PhysicalNamingStrategyStandardImpl {
        return PhysicalNamingStrategyStandardImpl()
    }

}
```

See `HibernateJpaAutoConfiguration` and `JpaBaseConfiguration` for more details.

18.8.7. Configure Hibernate Second-Level Caching

Hibernate second-level cache can be configured for a range of cache providers. Rather than configuring Hibernate to lookup the cache provider again, it is better to provide the one that is available in the context whenever possible.

To do this with JCache, first make sure that `org.hibernate.orm:hibernate-jcache` is available on the classpath. Then, add a `HibernatePropertiesCustomizer` bean as shown in the following example:

Java

```
import org.hibernate.cache.jcache.ConfigSettings;

import org.springframework.boot.autoconfigure.orm.jpa.HibernatePropertiesCustomizer;
import org.springframework.cache.jcache.JCacheCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyHibernateSecondLevelCacheConfiguration {

    @Bean
    public HibernatePropertiesCustomizer
hibernateSecondLevelCacheCustomizer(JCacheCacheManager cacheManager) {
        return (properties) -> properties.put(ConfigSettings.CACHE_MANAGER,
cacheManager.getCacheManager());
    }

}
```

Kotlin

```
import org.hibernate.cache.jcache.ConfigSettings
import org.springframework.boot.autoconfigure.orm.jpa.HibernatePropertiesCustomizer
import org.springframework.cache.jcache.JCacheCacheManager
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyHibernateSecondLevelCacheConfiguration {

    @Bean
    fun hibernateSecondLevelCacheCustomizer(cacheManager: JCacheCacheManager):
HibernatePropertiesCustomizer {
        return HibernatePropertiesCustomizer { properties ->
            properties[ConfigSettings.CACHE_MANAGER] = cacheManager.cacheManager
        }
    }

}
```

This customizer will configure Hibernate to use the same `CacheManager` as the one that the application uses. It is also possible to use separate `CacheManager` instances. For details, see the Hibernate user guide.

18.8.8. Use Dependency Injection in Hibernate Components

By default, Spring Boot registers a `BeanContainer` implementation that uses the `BeanFactory` so that

converters and entity listeners can use regular dependency injection.

You can disable or tune this behavior by registering a `HibernatePropertiesCustomizer` that removes or changes the `hibernate.resource.beans.container` property.

18.8.9. Use a Custom EntityManagerFactory

To take full control of the configuration of the `EntityManagerFactory`, you need to add a `@Bean` named 'entityManagerFactory'. Spring Boot auto-configuration switches off its entity manager in the presence of a bean of that type.

18.8.10. Using Multiple EntityManagerFactories

If you need to use JPA against multiple data sources, you likely need one `EntityManagerFactory` per data source. The `LocalContainerEntityManagerFactoryBean` from Spring ORM allows you to configure an `EntityManagerFactory` for your needs. You can also reuse `JpaProperties` to bind settings for each `EntityManagerFactory`, as shown in the following example:

Java

```
import javax.sql.DataSource;

import org.springframework.boot.autoconfigure.orm.jpa.JpaProperties;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;

@Configuration(proxyBeanMethods = false)
public class MyEntityManagerFactoryConfiguration {

    @Bean
    @ConfigurationProperties("app.jpa.first")
    public JpaProperties firstJpaProperties() {
        return new JpaProperties();
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean firstEntityManagerFactory(DataSource
firstDataSource,
            JpaProperties firstJpaProperties) {
        EntityManagerFactoryBuilder builder =
createEntityManagerFactoryBuilder(firstJpaProperties);
        return
builder.dataSource(firstDataSource).packages(Order.class).persistenceUnit("firstDs").b
uild();
    }

    private EntityManagerFactoryBuilder
createEntityManagerFactoryBuilder(JpaProperties jpaProperties) {
        JpaVendorAdapter jpaVendorAdapter = createJpaVendorAdapter(jpaProperties);
        return new EntityManagerFactoryBuilder(jpaVendorAdapter,
jpaProperties.getProperties(), null);
    }

    private JpaVendorAdapter createJpaVendorAdapter(JpaProperties jpaProperties) {
        // ... map JPA properties as needed
        return new HibernateJpaVendorAdapter();
    }

}
```

Kotlin

```
import javax.sql.DataSource

import org.springframework.boot.autoconfigure.orm.jpa.JpaProperties
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.orm.jpa.JpaVendorAdapter
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter

@Configuration(proxyBeanMethods = false)
class MyEntityManagerFactoryConfiguration {

    @Bean
    @ConfigurationProperties("app.jpa.first")
    fun firstJpaProperties(): JpaProperties {
        return JpaProperties()
    }

    @Bean
    fun firstEntityManagerFactory(
        firstDataSource: DataSource?,
        firstJpaProperties: JpaProperties
    ): LocalContainerEntityManagerFactoryBean {
        val builder = createEntityManagerFactoryBuilder(firstJpaProperties)
        return
builder.dataSource(firstDataSource).packages(Order::class.java).persistenceUnit("first
Ds").build()
    }

    private fun createEntityManagerFactoryBuilder(jpaProperties: JpaProperties):
EntityManagerFactoryBuilder {
        val jpaVendorAdapter = createJpaVendorAdapter(jpaProperties)
        return EntityManagerFactoryBuilder(jpaVendorAdapter, jpaProperties.properties,
null)
    }

    private fun createJpaVendorAdapter(jpaProperties: JpaProperties): JpaVendorAdapter
{
        // ... map JPA properties as needed
        return HibernateJpaVendorAdapter()
    }

}
```

The example above creates an `EntityManagerFactory` using a `DataSource` bean named `firstDataSource`. It scans entities located in the same package as `Order`. It is possible to map

additional JPA properties using the `app.first.jpa` namespace.

NOTE	When you create a bean for `LocalContainerEntityManagerFactoryBean` yourself, any customization that was applied during the creation of the auto-configured `LocalContainerEntityManagerFactoryBean` is lost. For example, in case of Hibernate, any properties under the `spring.jpa.hibernate` prefix will not be automatically applied to your `LocalContainerEntityManagerFactoryBean`. If you were relying on these properties for configuring things like the naming strategy or the DDL mode, you will need to explicitly configure that when creating the `LocalContainerEntityManagerFactoryBean` bean.

You should provide a similar configuration for any additional data sources for which you need JPA access. To complete the picture, you need to configure a `JpaTransactionManager` for each `EntityManagerFactory` as well. Alternatively, you might be able to use a JTA transaction manager that spans both.

If you use Spring Data, you need to configure `@EnableJpaRepositories` accordingly, as shown in the following examples:

Java

```
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@Configuration(proxyBeanMethods = false)
@EnableJpaRepositories(basePackageClasses = Order.class, entityManagerFactoryRef =
"firstEntityManagerFactory")
public class OrderConfiguration {

}
```

Kotlin

```
import org.springframework.context.annotation.Configuration
import org.springframework.data.jpa.repository.config.EnableJpaRepositories

@Configuration(proxyBeanMethods = false)
@EnableJpaRepositories(basePackageClasses = [Order::class], entityManagerFactoryRef =
"firstEntityManagerFactory")
class OrderConfiguration
```

Java

```
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@Configuration(proxyBeanMethods = false)
@EnableJpaRepositories(basePackageClasses = Customer.class, entityManagerFactoryRef =
"secondEntityManagerFactory")
public class CustomerConfiguration {

}
```

Kotlin

```
import org.springframework.context.annotation.Configuration
import org.springframework.data.jpa.repository.config.EnableJpaRepositories

@Configuration(proxyBeanMethods = false)
@EnableJpaRepositories(basePackageClasses = [Customer::class], entityManagerFactoryRef
= "secondEntityManagerFactory")
class CustomerConfiguration
```

18.8.11. Use a Traditional persistence.xml File

Spring Boot will not search for or use a `META-INF/persistence.xml` by default. If you prefer to use a traditional `persistence.xml`, you need to define your own `@Bean` of type `LocalEntityManagerFactoryBean` (with an ID of 'entityManagerFactory') and set the persistence unit name there.

See `JpaBaseConfiguration` for the default settings.

18.8.12. Use Spring Data JPA and Mongo Repositories

Spring Data JPA and Spring Data Mongo can both automatically create `Repository` implementations for you. If they are both present on the classpath, you might have to do some extra configuration to tell Spring Boot which repositories to create. The most explicit way to do that is to use the standard Spring Data `@EnableJpaRepositories` and `@EnableMongoRepositories` annotations and provide the location of your `Repository` interfaces.

There are also flags (`spring.data.*.repositories.enabled` and `spring.data.*.repositories.type`) that you can use to switch the auto-configured repositories on and off in external configuration. Doing so is useful, for instance, in case you want to switch off the Mongo repositories and still use the auto-configured `MongoTemplate`.

The same obstacle and the same features exist for other auto-configured Spring Data repository types (Elasticsearch, Redis, and others). To work with them, change the names of the annotations and flags accordingly.

18.8.13. Customize Spring Data's Web Support

Spring Data provides web support that simplifies the use of Spring Data repositories in a web application. Spring Boot provides properties in the `spring.data.web` namespace for customizing its configuration. Note that if you are using Spring Data REST, you must use the properties in the `spring.data.rest` namespace instead.

18.8.14. Expose Spring Data Repositories as REST Endpoint

Spring Data REST can expose the `Repository` implementations as REST endpoints for you, provided Spring MVC has been enabled for the application.

Spring Boot exposes a set of useful properties (from the `spring.data.rest` namespace) that customize the `RepositoryRestConfiguration`. If you need to provide additional customization, you should use a `RepositoryRestConfigurer` bean.

NOTE	If you do not specify any order on your custom `RepositoryRestConfigurer`, it runs after the one Spring Boot uses internally. If you need to specify an order, make sure it is higher than 0.

18.8.15. Configure a Component that is Used by JPA

If you want to configure a component that JPA uses, then you need to ensure that the component is initialized before JPA. When the component is auto-configured, Spring Boot takes care of this for you. For example, when Flyway is auto-configured, Hibernate is configured to depend upon Flyway so that Flyway has a chance to initialize the database before Hibernate tries to use it.

If you are configuring a component yourself, you can use an `EntityManagerFactoryDependsOnPostProcessor` subclass as a convenient way of setting up the necessary dependencies. For example, if you use Hibernate Search with Elasticsearch as its index manager, any `EntityManagerFactory` beans must be configured to depend on the `elasticsearchClient` bean, as shown in the following example:

Java

```
import jakarta.persistence.EntityManagerFactory;

import
org.springframework.boot.autoconfigure.orm.jpa.EntityManagerFactoryDependsOnPostProces
sor;
import org.springframework.stereotype.Component;

/**
 * {@link EntityManagerFactoryDependsOnPostProcessor} that ensures that
 * {@link EntityManagerFactory} beans depend on the {@code elasticsearchClient} bean.
 */
@Component
public class ElasticsearchEntityManagerFactoryDependsOnPostProcessor
        extends EntityManagerFactoryDependsOnPostProcessor {

    public ElasticsearchEntityManagerFactoryDependsOnPostProcessor() {
        super("elasticsearchClient");
    }

}
```

Kotlin

```
import
org.springframework.boot.autoconfigure.orm.jpa.EntityManagerFactoryDependsOnPostProces
sor
import org.springframework.stereotype.Component

@Component
class ElasticsearchEntityManagerFactoryDependsOnPostProcessor :
    EntityManagerFactoryDependsOnPostProcessor("elasticsearchClient")
```

18.8.16. Configure jOOQ with Two DataSources

If you need to use jOOQ with multiple data sources, you should create your own `DSLContext` for each one. See JooqAutoConfiguration for more details.

TIP In particular, `JooqExceptionTranslator` and `SpringTransactionProvider` can be reused to provide similar features to what the auto-configuration does with a single `DataSource`.

18.9. Database Initialization

An SQL database can be initialized in different ways depending on what your stack is. Of course, you can also do it manually, provided the database is a separate process. It is recommended to use a single mechanism for schema generation.

18.9.1. Initialize a Database Using JPA

JPA has features for DDL generation, and these can be set up to run on startup against the database. This is controlled through two external properties:

- `spring.jpa.generate-ddl` (boolean) switches the feature on and off and is vendor independent.
- `spring.jpa.hibernate.ddl-auto` (enum) is a Hibernate feature that controls the behavior in a more fine-grained way. This feature is described in more detail later in this guide.

18.9.2. Initialize a Database Using Hibernate

You can set `spring.jpa.hibernate.ddl-auto` explicitly and the standard Hibernate property values are `none`, `validate`, `update`, `create`, and `create-drop`. Spring Boot chooses a default value for you based on whether it thinks your database is embedded. It defaults to `create-drop` if no schema manager has been detected or `none` in all other cases. An embedded database is detected by looking at the `Connection` type and JDBC url. `hsqldb`, `h2`, and `derby` are candidates, and others are not. Be careful when switching from in-memory to a 'real' database that you do not make assumptions about the existence of the tables and data in the new platform. You either have to set `ddl-auto` explicitly or use one of the other mechanisms to initialize the database.

NOTE	You can output the schema creation by enabling the `org.hibernate.SQL` logger. This is done for you automatically if you enable the debug mode.

In addition, a file named `import.sql` in the root of the classpath is executed on startup if Hibernate creates the schema from scratch (that is, if the `ddl-auto` property is set to `create` or `create-drop`). This can be useful for demos and for testing if you are careful but is probably not something you want to be on the classpath in production. It is a Hibernate feature (and has nothing to do with Spring).

18.9.3. Initialize a Database Using Basic SQL Scripts

Spring Boot can automatically create the schema (DDL scripts) of your JDBC `DataSource` or R2DBC `ConnectionFactory` and initialize its data (DML scripts).

By default, it loads schema scripts from `optional:classpath*:schema.sql` and data scripts from `optional:classpath*:data.sql`. The locations of these schema and data scripts can customized using `spring.sql.init.schema-locations` and `spring.sql.init.data-locations` respectively. The `optional:` prefix means that the application will start when the files do not exist. To have the application fail to start when the files are absent, remove the `optional:` prefix.

In addition, Spring Boot processes the `optional:classpath*:schema-${platform}.sql` and `optional:classpath*:data-${platform}.sql` files (if present), where `${platform}` is the value of `spring.sql.init.platform`. This allows you to switch to database-specific scripts if necessary. For example, you might choose to set it to the vendor name of the database (`hsqldb`, `h2`, `oracle`, `mysql`, `postgresql`, and so on).

By default, SQL database initialization is only performed when using an embedded in-memory database. To always initialize an SQL database, irrespective of its type, set `spring.sql.init.mode` to `always`. Similarly, to disable initialization, set `spring.sql.init.mode` to `never`. By default, Spring Boot

enables the fail-fast feature of its script-based database initializer. This means that, if the scripts cause exceptions, the application fails to start. You can tune that behavior by setting `spring.sql.init.continue-on-error`.

Script-based `DataSource` initialization is performed, by default, before any JPA `EntityManagerFactory` beans are created. `schema.sql` can be used to create the schema for JPA-managed entities and `data.sql` can be used to populate it. While we do not recommend using multiple data source initialization technologies, if you want script-based `DataSource` initialization to be able to build upon the schema creation performed by Hibernate, set `spring.jpa.defer-datasource-initialization` to `true`. This will defer data source initialization until after any `EntityManagerFactory` beans have been created and initialized. `schema.sql` can then be used to make additions to any schema creation performed by Hibernate and `data.sql` can be used to populate it.

NOTE	The initialization scripts support `--` for single line comments and `/* */` for block comments. Other comment formats are not supported.

If you are using a Higher-level Database Migration Tool, like Flyway or Liquibase, you should use them alone to create and initialize the schema. Using the basic `schema.sql` and `data.sql` scripts alongside Flyway or Liquibase is not recommended and support will be removed in a future release.

If you need to initialize test data using a higher-level database migration tool, please see the sections about Flyway and Liquibase.

18.9.4. Initialize a Spring Batch Database

If you use Spring Batch, it comes pre-packaged with SQL initialization scripts for most popular database platforms. Spring Boot can detect your database type and execute those scripts on startup. If you use an embedded database, this happens by default. You can also enable it for any database type, as shown in the following example:

Properties

```
spring.batch.jdbc.initialize-schema=always
```

Yaml

```
spring:
  batch:
    jdbc:
      initialize-schema: "always"
```

You can also switch off the initialization explicitly by setting `spring.batch.jdbc.initialize-schema` to `never`.

18.9.5. Use a Higher-level Database Migration Tool

Spring Boot supports two higher-level migration tools: Flyway and Liquibase.

Execute Flyway Database Migrations on Startup

To automatically run Flyway database migrations on startup, add the `org.flywaydb:flyway-core` to your classpath.

Typically, migrations are scripts in the form `V<VERSION>__<NAME>.sql` (with `<VERSION>` an underscore-separated version, such as '1' or '2_1'). By default, they are in a directory called `classpath:db/migration`, but you can modify that location by setting `spring.flyway.locations`. This is a comma-separated list of one or more `classpath:` or `filesystem:` locations. For example, the following configuration would search for scripts in both the default classpath location and the `/opt/migration` directory:

Properties

```
spring.flyway.locations=classpath:db/migration,filesystem:/opt/migration
```

Yaml

```
spring:
  flyway:
    locations: "classpath:db/migration,filesystem:/opt/migration"
```

You can also add a special `{vendor}` placeholder to use vendor-specific scripts. Assume the following:

Properties

```
spring.flyway.locations=classpath:db/migration/{vendor}
```

Yaml

```
spring:
  flyway:
    locations: "classpath:db/migration/{vendor}"
```

Rather than using `db/migration`, the preceding configuration sets the directory to use according to the type of the database (such as `db/migration/mysql` for MySQL). The list of supported databases is available in `DatabaseDriver`.

Migrations can also be written in Java. Flyway will be auto-configured with any beans that implement `JavaMigration`.

`FlywayProperties` provides most of Flyway's settings and a small set of additional properties that can be used to disable the migrations or switch off the location checking. If you need more control over the configuration, consider registering a `FlywayConfigurationCustomizer` bean.

Spring Boot calls `Flyway.migrate()` to perform the database migration. If you would like more control, provide a `@Bean` that implements `FlywayMigrationStrategy`.

Flyway supports SQL and Java callbacks. To use SQL-based callbacks, place the callback scripts in the `classpath:db/migration` directory. To use Java-based callbacks, create one or more beans that implement `Callback`. Any such beans are automatically registered with `Flyway`. They can be ordered by using `@Order` or by implementing `Ordered`. Beans that implement the deprecated `FlywayCallback` interface can also be detected, however they cannot be used alongside `Callback` beans.

By default, Flyway autowires the (`@Primary`) `DataSource` in your context and uses that for migrations. If you like to use a different `DataSource`, you can create one and mark its `@Bean` as `@FlywayDataSource`. If you do so and want two data sources, remember to create another one and mark it as `@Primary`. Alternatively, you can use Flyway's native `DataSource` by setting `spring.flyway.[url,user,password]` in external properties. Setting either `spring.flyway.url` or `spring.flyway.user` is sufficient to cause Flyway to use its own `DataSource`. If any of the three properties has not been set, the value of its equivalent `spring.datasource` property will be used.

You can also use Flyway to provide data for specific scenarios. For example, you can place test-specific migrations in `src/test/resources` and they are run only when your application starts for testing. Also, you can use profile-specific configuration to customize `spring.flyway.locations` so that certain migrations run only when a particular profile is active. For example, in `application-dev.properties`, you might specify the following setting:

Properties

```
spring.flyway.locations=classpath:/db/migration,classpath:/dev/db/migration
```

Yaml

```
spring:
  flyway:
    locations: "classpath:/db/migration,classpath:/dev/db/migration"
```

With that setup, migrations in `dev/db/migration` run only when the `dev` profile is active.

Execute Liquibase Database Migrations on Startup

To automatically run Liquibase database migrations on startup, add the `org.liquibase:liquibase-core` to your classpath.

NOTE

When you add the `org.liquibase:liquibase-core` to your classpath, database migrations run by default for both during application startup and before your tests run. This behavior can be customized by using the `spring.liquibase.enabled` property, setting different values in the `main` and `test` configurations. It is not possible to use two different ways to initialize the database (for example Liquibase for application startup, JPA for test runs).

By default, the master change log is read from `db/changelog/db.changelog-master.yaml`, but you can change the location by setting `spring.liquibase.change-log`. In addition to YAML, Liquibase also supports JSON, XML, and SQL change log formats.

By default, Liquibase autowires the (`@Primary`) `DataSource` in your context and uses that for migrations. If you need to use a different `DataSource`, you can create one and mark its `@Bean` as `@LiquibaseDataSource`. If you do so and you want two data sources, remember to create another one and mark it as `@Primary`. Alternatively, you can use Liquibase's native `DataSource` by setting `spring.liquibase.[driver-class-name,url,user,password]` in external properties. Setting either `spring.liquibase.url` or `spring.liquibase.user` is sufficient to cause Liquibase to use its own `DataSource`. If any of the three properties has not been set, the value of its equivalent `spring.datasource` property will be used.

See `LiquibaseProperties` for details about available settings such as contexts, the default schema, and others.

Use Flyway for test-only migrations

If you want to create Flyway migrations which populate your test database, place them in `src/test/resources/db/migration`. A file named, for example, `src/test/resources/db/migration/V9999__test-data.sql` will be executed after your production migrations and only if you're running the tests. You can use this file to create the needed test data. This file will not be packaged in your uber jar or your container.

Use Liquibase for test-only migrations

If you want to create Liquibase migrations which populate your test database, you have to create a test changelog which also includes the production changelog.

First, you need to configure Liquibase to use a different changelog when running the tests. One way to do this is to create a Spring Boot `test` profile and put the Liquibase properties in there. For that, create a file named `src/test/resources/application-test.properties` and put the following property in there:

Properties

```
spring.liquibase.change-log=classpath:/db/changelog/db.changelog-test.yaml
```

Yaml

```
spring:
  liquibase:
    change-log: "classpath:/db/changelog/db.changelog-test.yaml"
```

This configures Liquibase to use a different changelog when running in the `test` profile.

Now create the changelog file at `src/test/resources/db/changelog/db.changelog-test.yaml`:

```
databaseChangeLog:
  - include:
      file: classpath:/db/changelog/db.changelog-master.yaml
  - changeSet:
      runOrder: "last"
      id: "test"
      changes:
        # Insert your changes here
```

This changelog will be used when the tests are run and it will not be packaged in your uber jar or your container. It includes the production changelog and then declares a new changeset, whose `runOrder: last` setting specifies that it runs after all the production changesets have been run. You can now use for example the insert changeset to insert data or the sql changeset to execute SQL directly.

The last thing to do is to configure Spring Boot to activate the `test` profile when running tests. To do this, you can add the `@ActiveProfiles("test")` annotation to your `@SpringBootTest` annotated test classes.

18.9.6. Depend Upon an Initialized Database

Database initialization is performed while the application is starting up as part of application context refresh. To allow an initialized database to be accessed during startup, beans that act as database initializers and beans that require that database to have been initialized are detected automatically. Beans whose initialization depends upon the database having been initialized are configured to depend upon those that initialize it. If, during startup, your application tries to access the database and it has not been initialized, you can configure additional detection of beans that initialize the database and require the database to have been initialized.

Detect a Database Initializer

Spring Boot will automatically detect beans of the following types that initialize an SQL database:

- `DataSourceScriptDatabaseInitializer`
- `EntityManagerFactory`
- `Flyway`
- `FlywayMigrationInitializer`
- `R2dbcScriptDatabaseInitializer`
- `SpringLiquibase`

If you are using a third-party starter for a database initialization library, it may provide a detector such that beans of other types are also detected automatically. To have other beans be detected, register an implementation of `DatabaseInitializerDetector` in `META-INF/spring.factories`.

Detect a Bean That Depends On Database Initialization

Spring Boot will automatically detect beans of the following types that depends upon database initialization:

- `AbstractEntityManagerFactoryBean` (unless `spring.jpa.defer-datasource-initialization` is set to `true`)
- `DSLContext` (jOOQ)
- `EntityManagerFactory` (unless `spring.jpa.defer-datasource-initialization` is set to `true`)
- `JdbcOperations`
- `NamedParameterJdbcOperations`

If you are using a third-party starter data access library, it may provide a detector such that beans of other types are also detected automatically. To have other beans be detected, register an implementation of `DependsOnDatabaseInitializationDetector` in `META-INF/spring.factories`. Alternatively, annotate the bean's class or its `@Bean` method with `@DependsOnDatabaseInitialization`.

18.10. NoSQL

Spring Boot offers a number of starters that support NoSQL technologies. This section answers questions that arise from using NoSQL with Spring Boot.

18.10.1. Use Jedis Instead of Lettuce

By default, the Spring Boot starter (`spring-boot-starter-data-redis`) uses Lettuce. You need to exclude that dependency and include the Jedis one instead. Spring Boot manages both of these dependencies so you can switch to Jedis without specifying a version.

The following example shows how to do so in Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <exclusions>
        <exclusion>
            <groupId>io.lettuce</groupId>
            <artifactId>lettuce-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```

The following example shows how to do so in Gradle:

```
dependencies {
    implementation('org.springframework.boot:spring-boot-starter-data-redis') {
        exclude group: 'io.lettuce', module: 'lettuce-core'
    }
    implementation 'redis.clients:jedis'
    // ...
}
```

18.11. Messaging

Spring Boot offers a number of starters to support messaging. This section answers questions that arise from using messaging with Spring Boot.

18.11.1. Disable Transacted JMS Session

If your JMS broker does not support transacted sessions, you have to disable the support of transactions altogether. If you create your own `JmsListenerContainerFactory`, there is nothing to do, since, by default it cannot be transacted. If you want to use the `DefaultJmsListenerContainerFactoryConfigurer` to reuse Spring Boot's default, you can disable transacted sessions, as follows:

Java

```
import jakarta.jms.ConnectionFactory;

import
org.springframework.boot.autoconfigure.jms.DefaultJmsListenerContainerFactoryConfigure
r;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;

@Configuration(proxyBeanMethods = false)
public class MyJmsConfiguration {

    @Bean
    public DefaultJmsListenerContainerFactory
jmsListenerContainerFactory(ConnectionFactory connectionFactory,
            DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory listenerFactory = new
DefaultJmsListenerContainerFactory();
        configurer.configure(listenerFactory, connectionFactory);
        listenerFactory.setTransactionManager(null);
        listenerFactory.setSessionTransacted(false);
        return listenerFactory;
    }

}
```

Kotlin

```
import jakarta.jms.ConnectionFactory
import
org.springframework.boot.autoconfigure.jms.DefaultJmsListenerContainerFactoryConfigure
r
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.jms.config.DefaultJmsListenerContainerFactory

@Configuration(proxyBeanMethods = false)
class MyJmsConfiguration {

    @Bean
    fun jmsListenerContainerFactory(connectionFactory: ConnectionFactory?,
            configurer: DefaultJmsListenerContainerFactoryConfigurer):
DefaultJmsListenerContainerFactory {
        val listenerFactory = DefaultJmsListenerContainerFactory()
        configurer.configure(listenerFactory, connectionFactory)
        listenerFactory.setTransactionManager(null)
        listenerFactory.setSessionTransacted(false)
        return listenerFactory
    }

}
```

The preceding example overrides the default factory, and it should be applied to any other factory that your application defines, if any.

18.12. Batch Applications

A number of questions often arise when people use Spring Batch from within a Spring Boot application. This section addresses those questions.

18.12.1. Specifying a Batch Data Source

By default, batch applications require a `DataSource` to store job details. Spring Batch expects a single `DataSource` by default. To have it use a `DataSource` other than the application's main `DataSource`, declare a `DataSource` bean, annotating its `@Bean` method with `@BatchDataSource`. If you do so and want two data sources, remember to mark the other one `@Primary`. To take greater control, add `@EnableBatchProcessing` to one of your `@Configuration` classes or extend `DefaultBatchConfiguration`. See the Javadoc of `@EnableBatchProcessing` and `DefaultBatchConfiguration` for more details.

For more info about Spring Batch, see the Spring Batch project page.

18.12.2. Running Spring Batch Jobs on Startup

Spring Batch auto-configuration is enabled by adding `spring-boot-starter-batch` to your application's classpath.

If a single `Job` is found in the application context, it is executed on startup (see `JobLauncherApplicationRunner` for details). If multiple `Job` beans are found, the job that should be executed must be specified using `spring.batch.job.name`.

To disable running a `Job` found in the application context, set the `spring.batch.job.enabled` to `false`.

See BatchAutoConfiguration for more details.

18.12.3. Running From the Command Line

Spring Boot converts any command line argument starting with `--` to a property to add to the `Environment`, see accessing command line properties. This should not be used to pass arguments to batch jobs. To specify batch arguments on the command line, use the regular format (that is without `--`), as shown in the following example:

```
$ java -jar myapp.jar someParameter=someValue anotherParameter=anotherValue
```

If you specify a property of the `Environment` on the command line, it is ignored by the job. Consider the following command:

```
$ java -jar myapp.jar --server.port=7070 someParameter=someValue
```

This provides only one argument to the batch job: `someParameter=someValue`.

18.12.4. Restarting a stopped or failed Job

To restart a failed `Job`, all parameters (identifying and non-identifying) must be re-specified on the command line. Non-identifying parameters are **not** copied from the previous execution. This allows them to be modified or removed.

NOTE	When you're using a custom `JobParametersIncrementer`, you have to gather all parameters managed by the incrementer to restart a failed execution.

18.12.5. Storing the Job Repository

Spring Batch requires a data store for the `Job` repository. If you use Spring Boot, you must use an actual database. Note that it can be an in-memory database, see Configuring a Job Repository.

18.13. Actuator

Spring Boot includes the Spring Boot Actuator. This section answers questions that often arise from its use.

18.13.1. Change the HTTP Port or Address of the Actuator Endpoints

In a standalone application, the Actuator HTTP port defaults to the same as the main HTTP port. To

make the application listen on a different port, set the external property: `management.server.port`. To listen on a completely different network address (such as when you have an internal network for management and an external one for user applications), you can also set `management.server.address` to a valid IP address to which the server is able to bind.

For more detail, see the `ManagementServerProperties` source code and "Customizing the Management Server Port" in the "Production-ready features" section.

18.13.2. Customize the 'whitelabel' Error Page

Spring Boot installs a 'whitelabel' error page that you see in a browser client if you encounter a server error (machine clients consuming JSON and other media types should see a sensible response with the right error code).

NOTE	Set `server.error.whitelabel.enabled=false` to switch the default error page off. Doing so restores the default of the servlet container that you are using. Note that Spring Boot still tries to resolve the error view, so you should probably add your own error page rather than disabling it completely.

Overriding the error page with your own depends on the templating technology that you use. For example, if you use Thymeleaf, you can add an `error.html` template. If you use FreeMarker, you can add an `error.ftlh` template. In general, you need a `View` that resolves with a name of `error` or a `@Controller` that handles the `/error` path. Unless you replaced some of the default configuration, you should find a `BeanNameViewResolver` in your `ApplicationContext`, so a `@Bean` named `error` would be one way of doing that. See `ErrorMvcAutoConfiguration` for more options.

See also the section on "Error Handling" for details of how to register handlers in the servlet container.

18.13.3. Customizing Sanitization

To take control over the sanitization, define a `SanitizingFunction` bean. The `SanitizableData` with which the function is called provides access to the key and value as well as the `PropertySource` from which they came. This allows you to, for example, sanitize every value that comes from a particular property source. Each `SanitizingFunction` is called in order until a function changes the value of the sanitizable data.

18.13.4. Map Health Indicators to Micrometer Metrics

Spring Boot health indicators return a `Status` type to indicate the overall system health. If you want to monitor or alert on levels of health for a particular application, you can export these statuses as metrics with Micrometer. By default, the status codes "UP", "DOWN", "OUT_OF_SERVICE" and "UNKNOWN" are used by Spring Boot. To export these, you will need to convert these states to some set of numbers so that they can be used with a Micrometer `Gauge`.

The following example shows one way to write such an exporter:

Java

```
import io.micrometer.core.instrument.Gauge;
import io.micrometer.core.instrument.MeterRegistry;

import org.springframework.boot.actuate.health.HealthEndpoint;
import org.springframework.boot.actuate.health.Status;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyHealthMetricsExportConfiguration {

    public MyHealthMetricsExportConfiguration(MeterRegistry registry, HealthEndpoint
healthEndpoint) {
        // This example presumes common tags (such as the app) are applied elsewhere
        Gauge.builder("health", healthEndpoint,
this::getStatusCode).strongReference(true).register(registry);
    }

    private int getStatusCode(HealthEndpoint health) {
        Status status = health.health().getStatus();
        if (Status.UP.equals(status)) {
            return 3;
        }
        if (Status.OUT_OF_SERVICE.equals(status)) {
            return 2;
        }
        if (Status.DOWN.equals(status)) {
            return 1;
        }
        return 0;
    }

}
```

Kotlin

```
import io.micrometer.core.instrument.Gauge
import io.micrometer.core.instrument.MeterRegistry
import org.springframework.boot.actuate.health.HealthEndpoint
import org.springframework.boot.actuate.health.Status
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyHealthMetricsExportConfiguration(registry: MeterRegistry, healthEndpoint:
HealthEndpoint) {

    init {
        // This example presumes common tags (such as the app) are applied elsewhere
        Gauge.builder("health", healthEndpoint) { health ->
            getStatusCode(health).toDouble()
        }.strongReference(true).register(registry)
    }

    private fun getStatusCode(health: HealthEndpoint): Int {
        return when (health.health().status) {
            Status.UP -> 3
            Status.OUT_OF_SERVICE -> 2
            Status.DOWN -> 1
            else -> 0
        }
    }

}
```

18.14. Security

This section addresses questions about security when working with Spring Boot, including questions that arise from using Spring Security with Spring Boot.

For more about Spring Security, see the Spring Security project page.

18.14.1. Switch off the Spring Boot Security Configuration

If you define a `@Configuration` with a `SecurityFilterChain` bean in your application, it switches off the default webapp security settings in Spring Boot.

18.14.2. Change the UserDetailsService and Add User Accounts

If you provide a `@Bean` of type `AuthenticationManager`, `AuthenticationProvider`, or `UserDetailsService`, the default `@Bean` for `InMemoryUserDetailsManager` is not created. This means you have the full feature set of Spring Security available (such as various authentication options).

The easiest way to add user accounts is to provide your own `UserDetailsService` bean.

18.14.3. Enable HTTPS When Running behind a Proxy Server

Ensuring that all your main endpoints are only available over HTTPS is an important chore for any application. If you use Tomcat as a servlet container, then Spring Boot adds Tomcat's own `RemoteIpValve` automatically if it detects some environment settings, and you should be able to rely on the `HttpServletRequest` to report whether it is secure or not (even downstream of a proxy server that handles the real SSL termination). The standard behavior is determined by the presence or absence of certain request headers (`x-forwarded-for` and `x-forwarded-proto`), whose names are conventional, so it should work with most front-end proxies. You can switch on the valve by adding some entries to `application.properties`, as shown in the following example:

Properties

```
server.tomcat.remoteip.remote-ip-header=x-forwarded-for
server.tomcat.remoteip.protocol-header=x-forwarded-proto
```

Yaml

```
server:
  tomcat:
    remoteip:
      remote-ip-header: "x-forwarded-for"
      protocol-header: "x-forwarded-proto"
```

(The presence of either of those properties switches on the valve. Alternatively, you can add the `RemoteIpValve` by customizing the `TomcatServletWebServerFactory` using a `WebServerFactoryCustomizer` bean.)

To configure Spring Security to require a secure channel for all (or some) requests, consider adding your own `SecurityFilterChain` bean that adds the following `HttpSecurity` configuration:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class MySecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        // Customize the application security ...
        http.requiresChannel((channel) -> channel.anyRequest().requiresSecure());
        return http.build();
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import org.springframework.security.web.SecurityFilterChain

@Configuration
class MySecurityConfig {

    @Bean
    fun securityFilterChain(http: HttpSecurity): SecurityFilterChain {
        // Customize the application security ...
        http.requiresChannel { requests -> requests.anyRequest().requiresSecure() }
        return http.build()
    }

}
```

18.15. Hot Swapping

Spring Boot supports hot swapping. This section answers questions about how it works.

18.15.1. Reload Static Content

There are several options for hot reloading. The recommended approach is to use `spring-boot-devtools`, as it provides additional development-time features, such as support for fast application restarts and LiveReload as well as sensible development-time configuration (such as template caching). Devtools works by monitoring the classpath for changes. This means that static resource

changes must be "built" for the change to take effect. By default, this happens automatically in Eclipse when you save your changes. In IntelliJ IDEA, the Make Project command triggers the necessary build. Due to the default restart exclusions, changes to static resources do not trigger a restart of your application. They do, however, trigger a live reload.

Alternatively, running in an IDE (especially with debugging on) is a good way to do development (all modern IDEs allow reloading of static resources and usually also allow hot-swapping of Java class changes).

Finally, the Maven and Gradle plugins can be configured (see the `addResources` property) to support running from the command line with reloading of static files directly from source. You can use that with an external css/js compiler process if you are writing that code with higher-level tools.

18.15.2. Reload Templates without Restarting the Container

Most of the templating technologies supported by Spring Boot include a configuration option to disable caching (described later in this document). If you use the `spring-boot-devtools` module, these properties are automatically configured for you at development time.

Thymeleaf Templates

If you use Thymeleaf, set `spring.thymeleaf.cache` to `false`. See `ThymeleafAutoConfiguration` for other Thymeleaf customization options.

FreeMarker Templates

If you use FreeMarker, set `spring.freemarker.cache` to `false`. See `FreeMarkerAutoConfiguration` for other FreeMarker customization options.

Groovy Templates

If you use Groovy templates, set `spring.groovy.template.cache` to `false`. See `GroovyTemplateAutoConfiguration` for other Groovy customization options.

18.15.3. Fast Application Restarts

The `spring-boot-devtools` module includes support for automatic application restarts. While not as fast as technologies such as JRebel it is usually significantly faster than a “cold start”. You should probably give it a try before investigating some of the more complex reload options discussed later in this document.

For more details, see the Developer Tools section.

18.15.4. Reload Java Classes without Restarting the Container

Many modern IDEs (Eclipse, IDEA, and others) support hot swapping of bytecode. Consequently, if you make a change that does not affect class or method signatures, it should reload cleanly with no side effects.

18.16. Testing

Spring Boot includes a number of testing utilities and support classes as well as a dedicated starter that provides common test dependencies. This section answers common questions about testing.

18.16.1. Testing With Spring Security

Spring Security provides support for running tests as a specific user. For example, the test in the snippet below will run with an authenticated user that has the `ADMIN` role.

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.security.test.context.support.WithMockUser;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

@WebMvcTest(UserController.class)
class MySecurityTests {

    @Autowired
    private MockMvc mvc;

    @Test
    @WithMockUser(roles = "ADMIN")
    void requestProtectedUrlWithUser() throws Exception {
        this.mvc.perform(get("/"));
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.security.test.context.support.WithMockUser
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders

@WebMvcTest(UserController::class)
class MySecurityTests(@Autowired val mvc: MockMvc) {

    @Test
    @WithMockUser(roles = ["ADMIN"])
    fun requestProtectedUrlWithUser() {
        mvc.perform(MockMvcRequestBuilders.get("/"))
    }

}
```

Spring Security provides comprehensive integration with Spring MVC Test and this can also be used when testing controllers using the `@WebMvcTest` slice and `MockMvc`.

For additional details on Spring Security's testing support, see Spring Security's reference documentation.

18.16.2. Structure `@Configuration` classes for inclusion in slice tests

Slice tests work by restricting Spring Framework's component scanning to a limited set of components based on their type. For any beans that are not created through component scanning, for example, beans that are created using the `@Bean` annotation, slice tests will not be able to include/exclude them from the application context. Consider this example:

```
import org.apache.commons.dbcp2.BasicDataSource;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration(proxyBeanMethods = false)
public class MyConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http.authorizeHttpRequests((requests) ->
requests.anyRequest().authenticated());
        return http.build();
    }

    @Bean
    @ConfigurationProperties("app.datasource.second")
    public BasicDataSource secondDataSource() {
        return DataSourceBuilder.create().type(BasicDataSource.class).build();
    }

}
```

For a `@WebMvcTest` for an application with the above `@Configuration` class, you might expect to have the `SecurityFilterChain` bean in the application context so that you can test if your controller endpoints are secured properly. However, `MyConfiguration` is not picked up by @WebMvcTest's component scanning filter because it doesn't match any of the types specified by the filter. You can include the configuration explicitly by annotating the test class with `@Import(MyConfiguration.class)`. This will load all the beans in `MyConfiguration` including the `BasicDataSource` bean which isn't required when testing the web tier. Splitting the configuration class into two will enable importing just the security configuration.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration(proxyBeanMethods = false)
public class MySecurityConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http.authorizeHttpRequests((requests) ->
requests.anyRequest().authenticated());
        return http.build();
    }

}
```

```
import org.apache.commons.dbcp2.BasicDataSource;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyDatasourceConfiguration {

    @Bean
    @ConfigurationProperties("app.datasource.second")
    public BasicDataSource secondDataSource() {
        return DataSourceBuilder.create().type(BasicDataSource.class).build();
    }

}
```

Having a single configuration class can be inefficient when beans of a certain domain need to be included in slice tests. Instead, structuring the application's configuration as multiple granular classes with beans for a specific domain can enable importing them only for specific slice tests.

18.17. Build

Spring Boot includes build plugins for Maven and Gradle. This section answers common questions about these plugins.

18.17.1. Generate Build Information

Both the Maven plugin and the Gradle plugin allow generating build information containing the coordinates, name, and version of the project. The plugins can also be configured to add additional properties through configuration. When such a file is present, Spring Boot auto-configures a `BuildProperties` bean.

To generate build information with Maven, add an execution for the `build-info` goal, as shown in the following example:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>3.1.6</version>
            <executions>
                <execution>
                    <goals>
                        <goal>build-info</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

TIP See the Spring Boot Maven Plugin documentation for more details.

The following example does the same with Gradle:

```
springBoot {
    buildInfo()
}
```

TIP See the Spring Boot Gradle Plugin documentation for more details.

18.17.2. Generate Git Information

Both Maven and Gradle allow generating a `git.properties` file containing information about the state of your `git` source code repository when the project was built.

For Maven users, the `spring-boot-starter-parent` POM includes a pre-configured plugin to generate a `git.properties` file. To use it, add the following declaration for the `Git Commit Id Plugin` to your POM:

```
<build>
    <plugins>
        <plugin>
            <groupId>io.github.git-commit-id</groupId>
            <artifactId>git-commit-id-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Gradle users can achieve the same result by using the `gradle-git-properties` plugin, as shown in the following example:

```
plugins {
    id "com.gorylenko.gradle-git-properties" version "2.4.1"
}
```

Both the Maven and Gradle plugins allow the properties that are included in `git.properties` to be configured.

TIP The commit time in `git.properties` is expected to match the following format: `yyyy-MM-dd'T'HH:mm:ssZ`. This is the default format for both plugins listed above. Using this format lets the time be parsed into a `Date` and its format, when serialized to JSON, to be controlled by Jackson's date serialization configuration settings.

18.17.3. Customize Dependency Versions

The `spring-boot-dependencies` POM manages the versions of common dependencies. The Spring Boot plugins for Maven and Gradle allow these managed dependency versions to be customized using build properties.

WARNING Each Spring Boot release is designed and tested against this specific set of third-party dependencies. Overriding versions may cause compatibility issues.

To override dependency versions with Maven, see this section of the Maven plugin's documentation.

To override dependency versions in Gradle, see this section of the Gradle plugin's documentation.

18.17.4. Create an Executable JAR with Maven

The `spring-boot-maven-plugin` can be used to create an executable "fat" JAR. If you use the `spring-boot-starter-parent` POM, you can declare the plugin and your jars are repackaged as follows:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

If you do not use the parent POM, you can still use the plugin. However, you must additionally add an `<executions>` section, as follows:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>3.1.6</version>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

See the plugin documentation for full usage details.

18.17.5. Use a Spring Boot Application as a Dependency

Like a war file, a Spring Boot application is not intended to be used as a dependency. If your application contains classes that you want to share with other projects, the recommended approach is to move that code into a separate module. The separate module can then be depended upon by your application and other projects.

If you cannot rearrange your code as recommended above, Spring Boot's Maven and Gradle plugins must be configured to produce a separate artifact that is suitable for use as a dependency. The executable archive cannot be used as a dependency as the executable jar format packages application classes in `BOOT-INF/classes`. This means that they cannot be found when the executable jar is used as a dependency.

To produce the two artifacts, one that can be used as a dependency and one that is executable, a classifier must be specified. This classifier is applied to the name of the executable archive, leaving the default archive for use as a dependency.

To configure a classifier of `exec` in Maven, you can use the following configuration:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <classifier>exec</classifier>
            </configuration>
        </plugin>
    </plugins>
</build>
```

18.17.6. Extract Specific Libraries When an Executable Jar Runs

Most nested libraries in an executable jar do not need to be unpacked in order to run. However, certain libraries can have problems. For example, JRuby includes its own nested jar support, which assumes that the `jruby-complete.jar` is always directly available as a file in its own right.

To deal with any problematic libraries, you can flag that specific nested jars should be automatically unpacked when the executable jar first runs. Such nested jars are written beneath the temporary directory identified by the `java.io.tmpdir` system property.

WARNING	Care should be taken to ensure that your operating system is configured so that it will not delete the jars that have been unpacked to the temporary directory while the application is still running.

For example, to indicate that JRuby should be flagged for unpacking by using the Maven Plugin, you would add the following configuration:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <requiresUnpack>
                    <dependency>
                        <groupId>org.jruby</groupId>
                        <artifactId>jruby-complete</artifactId>
                    </dependency>
                </requiresUnpack>
            </configuration>
        </plugin>
    </plugins>
</build>
```

18.17.7. Create a Non-executable JAR with Exclusions

Often, if you have an executable and a non-executable jar as two separate build products, the executable version has additional configuration files that are not needed in a library jar. For example, the `application.yaml` configuration file might be excluded from the non-executable JAR.

In Maven, the executable jar must be the main artifact and you can add a classified jar for the library, as follows:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <executions>
                <execution>
                    <id>lib</id>
                    <phase>package</phase>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                    <configuration>
                        <classifier>lib</classifier>
                        <excludes>
                            <exclude>application.yaml</exclude>
                        </excludes>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

18.17.8. Remote Debug a Spring Boot Application Started with Maven

To attach a remote debugger to a Spring Boot application that was started with Maven, you can use the `jvmArguments` property of the maven plugin.

See this example for more details.

18.17.9. Build an Executable Archive From Ant without Using spring-boot-antlib

To build with Ant, you need to grab dependencies, compile, and then create a jar or war archive. To make it executable, you can either use the `spring-boot-antlib` module or you can follow these instructions:

1. If you are building a jar, package the application's classes and resources in a nested `BOOT-INF/classes` directory. If you are building a war, package the application's classes in a nested `WEB-INF/classes` directory as usual.
2. Add the runtime dependencies in a nested `BOOT-INF/lib` directory for a jar or `WEB-INF/lib` for a war. Remember **not** to compress the entries in the archive.
3. Add the `provided` (embedded container) dependencies in a nested `BOOT-INF/lib` directory for a jar or `WEB-INF/lib-provided` for a war. Remember **not** to compress the entries in the archive.
4. Add the `spring-boot-loader` classes at the root of the archive (so that the `Main-Class` is available).
5. Use the appropriate launcher (such as `JarLauncher` for a jar file) as a `Main-Class` attribute in the manifest and specify the other properties it needs as manifest entries — principally, by setting a `Start-Class` property.

The following example shows how to build an executable archive with Ant:

```
<target name="build" depends="compile">
    <jar destfile="target/${ant.project.name}-${spring-boot.version}.jar"
compress="false">
        <mappedresources>
            <fileset dir="target/classes" />
            <globmapper from="*" to="BOOT-INF/classes/*"/>
        </mappedresources>
        <mappedresources>
            <fileset dir="src/main/resources" erroronmissingdir="false"/>
            <globmapper from="*" to="BOOT-INF/classes/*"/>
        </mappedresources>
        <mappedresources>
            <fileset dir="${lib.dir}/runtime" />
            <globmapper from="*" to="BOOT-INF/lib/*"/>
        </mappedresources>
        <zipfileset src="${lib.dir}/loader/spring-boot-loader-jar-${spring-
boot.version}.jar" />
        <manifest>
            <attribute name="Main-Class"
value="org.springframework.boot.loader.JarLauncher" />
            <attribute name="Start-Class" value="${start-class}" />
        </manifest>
    </jar>
</target>
```

18.18. Ahead-of-time processing

A number of questions often arise when people use the ahead-of-time processing of Spring Boot applications. This section addresses those questions.

18.18.1. Conditions

Ahead-of-time processing optimizes the application and evaluates conditions based on the environment at build time. Profiles are implemented through conditions and are therefore affected, too.

If you want beans that are created based on a condition in an ahead-of-time optimized application, you have to set up the environment when building the application. The beans which are created while ahead-of-time processing at build time are then always created when running the application and can't be switched off. To do this, you can set the profiles which should be used when building the application.

For Maven, this works by setting the `profiles` configuration of the `spring-boot-maven-plugin:process-aot` execution:

```
<profile>
    <id>native</id>
    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                    <executions>
                        <execution>
                            <id>process-aot</id>
                            <configuration>
                                <profiles>profile-a,profile-b</profiles>
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
</profile>
```

For Gradle, you need to configure the `ProcessAot` task:

```
tasks.withType(org.springframework.boot.gradle.tasks.aot.ProcessAot).configureEach {
    args('--spring.profiles.active=profile-a,profile-b')
}
```

Profiles which only change configuration properties that don't influence conditions are supported without limitations when running ahead-of-time optimized applications.

18.19. Traditional Deployment

Spring Boot supports traditional deployment as well as more modern forms of deployment. This section answers common questions about traditional deployment.

18.19.1. Create a Deployable War File

WARNING	Because Spring WebFlux does not strictly depend on the servlet API and applications are deployed by default on an embedded Reactor Netty server, War deployment is not supported for WebFlux applications.

The first step in producing a deployable war file is to provide a `SpringBootServletInitializer` subclass and override its `configure` method. Doing so makes use of Spring Framework's servlet 3.0 support and lets you configure your application when it is launched by the servlet container. Typically, you should update your application's main class to extend `SpringBootServletInitializer`, as shown in the following example:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)
{
        return application.sources(MyApplication.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.builder.SpringApplicationBuilder
import org.springframework.boot.runApplication
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer

@SpringBootApplication
class MyApplication : SpringBootServletInitializer() {

    override fun configure(application: SpringApplicationBuilder):
SpringApplicationBuilder {
        return application.sources(MyApplication::class.java)
    }

}

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

The next step is to update your build configuration such that your project produces a war file rather than a jar file. If you use Maven and `spring-boot-starter-parent` (which configures Maven's war plugin for you), all you need to do is to modify `pom.xml` to change the packaging to war, as follows:

```
<packaging>war</packaging>
```

If you use Gradle, you need to modify `build.gradle` to apply the war plugin to the project, as follows:

```
apply plugin: 'war'
```

The final step in the process is to ensure that the embedded servlet container does not interfere with the servlet container to which the war file is deployed. To do so, you need to mark the embedded servlet container dependency as being provided.

If you use Maven, the following example marks the servlet container (Tomcat, in this case) as being provided:

```
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <!-- ... -->
</dependencies>
```

If you use Gradle, the following example marks the servlet container (Tomcat, in this case) as being provided:

```
dependencies {
    // ...
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    // ...
}
```

TIP `providedRuntime` is preferred to Gradle's `compileOnly` configuration. Among other limitations, `compileOnly` dependencies are not on the test classpath, so any web-based integration tests fail.

If you use the Spring Boot build tools, marking the embedded servlet container dependency as provided produces an executable war file with the provided dependencies packaged in a `lib-provided` directory. This means that, in addition to being deployable to a servlet container, you can also run your application by using `java -jar` on the command line.

18.19.2. Convert an Existing Application to Spring Boot

To convert an existing non-web Spring application to a Spring Boot application, replace the code that creates your `ApplicationContext` and replace it with calls to `SpringApplication` or `SpringApplicationBuilder`. Spring MVC web applications are generally amenable to first creating a deployable war application and then migrating it later to an executable war or jar. See the Getting Started Guide on Converting a jar to a war.

To create a deployable war by extending `SpringBootServletInitializer` (for example, in a class called `Application`) and adding the Spring Boot `@SpringBootApplication` annotation, use code similar to that shown in the following example:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)
{
        // Customize the application or call application.sources(...) to add sources
        // Since our example is itself a @Configuration class (through
        // @SpringBootApplication)
        // we actually do not need to override this method.
        return application;
    }


}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.builder.SpringApplicationBuilder
import org.springframework.boot.runApplication
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer

@SpringBootApplication
class MyApplication : SpringBootServletInitializer() {

    override fun configure(application: SpringApplicationBuilder):
SpringApplicationBuilder {
        // Customize the application or call application.sources(...) to add sources
        // Since our example is itself a @Configuration class (through
@SpringBootApplication)
        // we actually do not need to override this method.
        return application
    }

}
```

Remember that, whatever you put in the `sources` is merely a Spring `ApplicationContext`. Normally, anything that already works should work here. There might be some beans you can remove later and let Spring Boot provide its own defaults for them, but it should be possible to get something working before you need to do that.

Static resources can be moved to `/public` (or `/static` or `/resources` or `/META-INF/resources`) in the

classpath root. The same applies to `messages.properties` (which Spring Boot automatically detects in the root of the classpath).

Vanilla usage of Spring `DispatcherServlet` and Spring Security should require no further changes. If you have other features in your application (for instance, using other servlets or filters), you may need to add some configuration to your `Application` context, by replacing those elements from the `web.xml`, as follows:

- A `@Bean` of type `Servlet` or `ServletRegistrationBean` installs that bean in the container as if it were a `<servlet/>` and `<servlet-mapping/>` in `web.xml`.
- A `@Bean` of type `Filter` or `FilterRegistrationBean` behaves similarly (as a `<filter/>` and `<filter-mapping/>`).
- An `ApplicationContext` in an XML file can be added through an `@ImportResource` in your `Application`. Alternatively, cases where annotation configuration is heavily used already can be recreated in a few lines as `@Bean` definitions.

Once the war file is working, you can make it executable by adding a `main` method to your `Application`, as shown in the following example:

Java

```
public static void main(String[] args) {
    SpringApplication.run(MyApplication.class, args);
}
```

Kotlin

```
fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

NOTE

If you intend to start your application as a war or as an executable application, you need to share the customizations of the builder in a method that is both available to the `SpringBootServletInitializer` callback and in the `main` method in a class similar to the following:

Java

```
import org.springframework.boot.Banner;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import
org.springframework.boot.web.servlet.support.SpringBootServletInitialize
r;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder
configure(SpringApplicationBuilder builder) {
        return customizerBuilder(builder);
    }

    public static void main(String[] args) {
        customizerBuilder(new SpringApplicationBuilder()).run(args);
    }

    private static SpringApplicationBuilder
customizerBuilder(SpringApplicationBuilder builder) {
        return
builder.sources(MyApplication.class).bannerMode(Banner.Mode.OFF);
    }

}
```

Kotlin

```
import org.springframework.boot.Banner
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.builder.SpringApplicationBuilder
import
org.springframework.boot.web.servlet.support.SpringBootServletInitialize
r

@SpringBootApplication
class MyApplication : SpringBootServletInitializer() {

    override fun configure(builder: SpringApplicationBuilder):
SpringApplicationBuilder {
        return customizerBuilder(builder)
    }

    companion object {

        @JvmStatic
        fun main(args: Array<String>) {
            customizerBuilder(SpringApplicationBuilder()).run(*args)
        }

        private fun customizerBuilder(builder:
SpringApplicationBuilder): SpringApplicationBuilder {
            return
builder.sources(MyApplication::class.java).bannerMode(Banner.Mode.OFF)
        }

    }

}
```

Applications can fall into more than one category:

- Servlet 3.0+ applications with no `web.xml`.
- Applications with a `web.xml`.
- Applications with a context hierarchy.
- Applications without a context hierarchy.

All of these should be amenable to translation, but each might require slightly different techniques.

Servlet 3.0+ applications might translate pretty easily if they already use the Spring Servlet 3.0+ initializer support classes. Normally, all the code from an existing `WebApplicationInitializer` can be moved into a `SpringBootServletInitializer`. If your existing application has more than one `ApplicationContext` (for example, if it uses `AbstractDispatcherServletInitializer`) then you might be able to combine all your context sources into a single `SpringApplication`. The main complication you might encounter is if combining does not work and you need to maintain the context hierarchy. See

the entry on building a hierarchy for examples. An existing parent context that contains web-specific features usually needs to be broken up so that all the `ServletContextAware` components are in the child context.

Applications that are not already Spring applications might be convertible to Spring Boot applications, and the previously mentioned guidance may help. However, you may yet encounter problems. In that case, we suggest asking questions on Stack Overflow with a tag of `spring-boot`.

18.19.3. Deploying a WAR to WebLogic

To deploy a Spring Boot application to WebLogic, you must ensure that your servlet initializer **directly** implements `WebApplicationInitializer` (even if you extend from a base class that already implements it).

A typical initializer for WebLogic should resemble the following example:

Java

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
import org.springframework.web.WebApplicationInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer implements
WebApplicationInitializer {

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer
import org.springframework.web.WebApplicationInitializer

@SpringBootApplication
class MyApplication : SpringBootServletInitializer(), WebApplicationInitializer
```

If you use Logback, you also need to tell WebLogic to prefer the packaged version rather than the version that was pre-installed with the server. You can do so by adding a `WEB-INF/weblogic.xml` file with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<wls:weblogic-web-app
    xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-web-app"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        https://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
        http://xmlns.oracle.com/weblogic/weblogic-web-app
        https://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd">
    <wls:container-descriptor>
        <wls:prefer-application-packages>
            <wls:package-name>org.slf4j</wls:package-name>
        </wls:prefer-application-packages>
    </wls:container-descriptor>
</wls:weblogic-web-app>
```

18.20. Docker Compose

This section includes topics relating to the Docker Compose support in Spring Boot.

18.20.1. Customizing the JDBC URL

When using `JdbcConnectionDetails` with Docker Compose, the parameters of the JDBC URL can be customized by applying the `org.springframework.boot.jdbc.parameters` label to the service. For example:

```
services:
  postgres:
    image: 'postgres:15.3'
    environment:
      - 'POSTGRES_USER=myuser'
      - 'POSTGRES_PASSWORD=secret'
      - 'POSTGRES_DB=mydb'
    ports:
      - '5432:5432'
    labels:
      org.springframework.boot.jdbc.parameters: 'ssl=true&sslmode=require'
```

With this Docker Compose file in place, the JDBC URL used is `jdbc:postgresql://127.0.0.1:5432/mydb?ssl=true&sslmode=require`.

18.20.2. Sharing services between multiple applications

If you want to share services between multiple applications, create the `compose.yaml` file in one of the applications and then use the configuration property `spring.docker.compose.file` in the other applications to reference the `compose.yaml` file. You should also set `spring.docker.compose.lifecycle-management` to `start-only`, as it defaults to `start-and-stop` and stopping one application would shut down the shared services for the other still running applications, too. Setting it to `start-only` won't

stop the shared services on application stop, but a caveat is that if you shut down all applications, the services stay running. You can stop the services manually by running `docker compose stop` on the commandline in the directory which contains the `compose.yaml` file.

Appendices

Appendix A: Common Application Properties

Various properties can be specified inside your `application.properties` file, inside your `application.yaml` file, or as command line switches. This appendix provides a list of common Spring Boot properties and references to the underlying classes that consume them.

TIP Spring Boot provides various conversion mechanism with advanced value formatting, make sure to review the properties conversion section.

NOTE Property contributions can come from additional jar files on your classpath, so you should not consider this an exhaustive list. Also, you can define your own properties.

.A.1. Core Properties

Name	Description	Default Value
`debug`	Enable debug logs.	`false`
`info.*`	Arbitrary properties to add to the info endpoint.	
`logging.charset.console`	Charset to use for console output.	
`logging.charset.file`	Charset to use for file output.	
`logging.config`	Location of the logging configuration file. For instance, `classpath:logback.xml` for Logback.	
`logging.exception-conversion-word`	Conversion word used when logging exceptions.	`%wEx`
`logging.file.name`	Log file name (for instance, `myapp.log`). Names can be an exact location or relative to the current directory.	
`logging.file.path`	Location of the log file. For instance, `/var/log`.	

Name	Description	Default Value
`logging.group.*`	Log groups to quickly change multiple loggers at the same time. For instance, `logging.group.db=org.hibernate,org.springframework.jdbc`.	
`logging.level.*`	Log levels severity mapping. For instance, `logging.level.org.springframework=DEBUG`.	
`logging.log4j2.config.override`	Overriding configuration files used to create a composite configuration.	
`logging.logback.rollingpolicy.clean-history-on-start`	Whether to clean the archive log files on startup.	`false`
`logging.logback.rollingpolicy.file-name-pattern`	Pattern for rolled-over log file names.	`${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz`
`logging.logback.rollingpolicy.max-file-size`	Maximum log file size.	`10MB`
`logging.logback.rollingpolicy.max-history`	Maximum number of archive log files to keep.	`7`
`logging.logback.rollingpolicy.total-size-cap`	Total size of log backups to be kept.	`0B`
`logging.pattern.console`	Appender pattern for output to the console. Supported only with the default Logback setup.	`%clr(%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p}) %clr(${PID:-}){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}`
`logging.pattern.dateformat`	Appender pattern for log date format. Supported only with the default Logback setup.	`yyyy-MM-dd'T'HH:mm:ss.SSSXXX`

Name	Description	Default Value
logging.pattern.file	Appender pattern for output to a file. Supported only with the default Logback setup.	%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}} ${LOG_LEVEL_PATTERN:-%5p} ${PID:- } --- [%t] %-40.40logger{39} : %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}
logging.pattern.level	Appender pattern for log level. Supported only with the default Logback setup.	%5p
logging.register-shutdown-hook	Register a shutdown hook for the logging system when it is initialized. Disabled automatically when deployed as a war file.	true
logging.threshold.console	Log level threshold for console output.	TRACE
logging.threshold.file	Log level threshold for file output.	TRACE
spring.aop.auto	Add @EnableAspectJAutoProxy.	true
spring.aop.proxy-target-class	Whether subclass-based (CGLIB) proxies are to be created (true), as opposed to standard Java interface-based proxies (false).	true
spring.application.admin.enabled	Whether to enable admin features for the application.	false
spring.application.admin.jmx-name	JMX name of the application admin MBean.	org.springframework.boot:type=Admin,name=SpringApplication
spring.application.name	Application name.	
spring.autoconfigure.exclude	Auto-configuration classes to exclude.	
spring.banner.charset	Banner file encoding.	UTF-8
spring.banner.location	Banner text resource location.	classpath:banner.txt
spring.beaninfo.ignore	Whether to skip search of BeanInfo classes.	true

Name	Description	Default Value
spring.codec.log-request-details	Whether to log form data at DEBUG level, and headers at TRACE level.	false
spring.codec.max-in-memory-size	Limit on the number of bytes that can be buffered whenever the input stream needs to be aggregated. This applies only to the auto-configured WebFlux server and WebClient instances. By default this is not set, in which case individual codec defaults apply. Most codecs are limited to 256K by default.	
spring.config.activate.on-cloud-platform	Required cloud platform for the document to be included.	
spring.config.activate.on-profile	Profile expressions that should match for the document to be included.	
spring.config.additional-location	Config file locations used in addition to the defaults.	
spring.config.import	Import additional config data.	
spring.config.location	Config file locations that replace the defaults.	
spring.config.name	Config file name.	application
spring.info.build.encoding	File encoding.	UTF-8
spring.info.build.location	Location of the generated build-info.properties file.	classpath:META-INF/build-info.properties
spring.info.git.encoding	File encoding.	UTF-8
spring.info.git.location	Location of the generated git.properties file.	classpath:git.properties
spring.jmx.default-domain	JMX domain name.	
spring.jmx.enabled	Expose management beans to the JMX domain.	false
spring.jmx.registration-policy	JMX Registration policy.	fail-on-existing
spring.jmx.server	MBeanServer bean name.	mbeanServer

Name	Description	Default Value
spring.jmx.unique-names	Whether unique runtime object names should be ensured.	false
spring.lifecycle.timeout-per-shutdown-phase	Timeout for the shutdown of any phase (group of SmartLifecycle beans with the same 'phase' value).	30s
spring.main.allow-bean-definition-overriding	Whether bean definition overriding, by registering a definition with the same name as an existing definition, is allowed.	false
spring.main.allow-circular-references	Whether to allow circular references between beans and automatically try to resolve them.	false
spring.main.banner-mode	Mode used to display the banner when the application runs.	console
spring.main.cloud-platform	Override the Cloud Platform auto-detection.	
spring.main.lazy-initialization	Whether initialization should be performed lazily.	false
spring.main.log-startup-info	Whether to log information about the application when it starts.	true
spring.main.register-shutdown-hook	Whether the application should have a shutdown hook registered.	true
spring.main.sources	Sources (class names, package names, or XML resource locations) to include in the ApplicationContext.	
spring.main.web-application-type	Flag to explicitly request a specific type of web application. If not set, auto-detected based on the classpath.	
spring.mandatory-file-encoding	Expected character encoding the application must use.	

Name	Description	Default Value
spring.messages.always-use-message-format	Whether to always apply the MessageFormat rules, parsing even messages without arguments.	false
spring.messages.basename	Comma-separated list of basenames (essentially a fully-qualified classpath location), each following the ResourceBundle convention with relaxed support for slash based locations. If it doesn't contain a package qualifier (such as "org.mypackage"), it will be resolved from the classpath root.	messages
spring.messages.cache-duration	Loaded resource bundle files cache duration. When not set, bundles are cached forever. If a duration suffix is not specified, seconds will be used.	
spring.messages.encoding	Message bundles encoding.	UTF-8
spring.messages.fallback-to-system-locale	Whether to fall back to the system Locale if no files for a specific Locale have been found. if this is turned off, the only fallback will be the default file (e.g. "messages.properties" for basename "messages").	true
spring.messages.use-code-as-default-message	Whether to use the message code as the default message instead of throwing a "NoSuchMessageException". Recommended during development only.	false
spring.output.ansi.enabled	Configures the ANSI output.	detect
spring.pid.fail-on-write-error	Fails if ApplicationPidFileWriter is used but it cannot write the PID file.	

Name	Description	Default Value
spring.pid.file	Location of the PID file to write (if ApplicationPidFileWriter is used).	
spring.profiles.active	Comma-separated list of active profiles. Can be overridden by a command line switch.	
spring.profiles.default	Name of the profile to enable if no profile is active.	default
spring.profiles.group.*	Profile groups to define a logical name for a related group of profiles.	
spring.profiles.include	Unconditionally activate the specified comma-separated list of profiles (or list of profiles if using YAML).	
spring.quartz.auto-startup	Whether to automatically start the scheduler after initialization.	true
spring.quartz.jdbc.comment-prefix	Prefixes for single-line comments in SQL initialization scripts.	[#, --]
spring.quartz.jdbc.initialize-schema	Database schema initialization mode.	embedded
spring.quartz.jdbc.platform	Platform to use in initialization scripts if the @@platform@@ placeholder is used. Auto-detected by default.	
spring.quartz.jdbc.schema	Path to the SQL file to use to initialize the database schema.	classpath:org/quartz/impl/jdbcjobstore/tables_@@platform@@.sql
spring.quartz.job-store-type	Quartz job store type.	memory
spring.quartz.overwrite-existing-jobs	Whether configured jobs should overwrite existing job definitions.	false
spring.quartz.properties.*	Additional Quartz Scheduler properties.	
spring.quartz.scheduler-name	Name of the scheduler.	quartzScheduler

Name	Description	Default Value
`spring.quartz.startup-delay`	Delay after which the scheduler is started once initialization completes. Setting this property makes sense if no jobs should be run before the entire application has started up.	`0s`
`spring.quartz.wait-for-jobs-to-complete-on-shutdown`	Whether to wait for running jobs to complete on shutdown.	`false`
`spring.reactor.debug-agent.enabled`	Whether the Reactor Debug Agent should be enabled when reactor-tools is present.	`true`
`spring.reactor.netty.shutdown-quiet-period`	Amount of time to wait before shutting down resources.	
`spring.ssl.bundle.jks.*`	Java keystore SSL trust material.	
`spring.ssl.bundle.pem.*`	PEM-encoded SSL trust material.	
`spring.task.execution.pool.allow-core-thread-timeout`	Whether core threads are allowed to time out. This enables dynamic growing and shrinking of the pool.	`true`
`spring.task.execution.pool.core-size`	Core number of threads.	`8`
`spring.task.execution.pool.keep-alive`	Time limit for which threads may remain idle before being terminated.	`60s`
`spring.task.execution.pool.max-size`	Maximum allowed number of threads. If tasks are filling up the queue, the pool can expand up to that size to accommodate the load. Ignored if the queue is unbounded.	
`spring.task.execution.pool.queue-capacity`	Queue capacity. An unbounded capacity does not increase the pool and therefore ignores the "max-size" property.	

Name	Description	Default Value
spring.task.execution.shutdown.await-termination	Whether the executor should wait for scheduled tasks to complete on shutdown.	false
spring.task.execution.shutdown.await-termination-period	Maximum time the executor should wait for remaining tasks to complete.	
spring.task.execution.thread-name-prefix	Prefix to use for the names of newly created threads.	task-
spring.task.scheduling.pool.size	Maximum allowed number of threads.	1
spring.task.scheduling.shutdown.await-termination	Whether the executor should wait for scheduled tasks to complete on shutdown.	false
spring.task.scheduling.shutdown.await-termination-period	Maximum time the executor should wait for remaining tasks to complete.	
spring.task.scheduling.thread-name-prefix	Prefix to use for the names of newly created threads.	scheduling-
trace	Enable trace logs.	false

.A.2. Cache Properties

Name	Description	Default Value
spring.cache.cache-names	Comma-separated list of cache names to create if supported by the underlying cache manager. Usually, this disables the ability to create additional caches on-the-fly.	
spring.cache.caffeine.spec	The spec to use to create caches. See CaffeineSpec for more details on the spec format.	
spring.cache.couchbase.expiration	Entry expiration. By default the entries never expire. Note that this value is ultimately converted to seconds.	

Name	Description	Default Value
spring.cache.infinispan.config	The location of the configuration file to use to initialize Infinispan.	
spring.cache.jcache.config	The location of the configuration file to use to initialize the cache manager. The configuration file is dependent of the underlying cache implementation.	
spring.cache.jcache.provider	Fully qualified name of the CachingProvider implementation to use to retrieve the JSR-107 compliant cache manager. Needed only if more than one JSR-107 implementation is available on the classpath.	
spring.cache.redis.cache-null-values	Allow caching null values.	true
spring.cache.redis.enable-statistics	Whether to enable cache statistics.	false
spring.cache.redis.key-prefix	Key prefix.	
spring.cache.redis.time-to-live	Entry expiration. By default the entries never expire.	
spring.cache.redis.use-key-prefix	Whether to use the key prefix when writing to Redis.	true
spring.cache.type	Cache type. By default, auto-detected according to the environment.	

.A.3. Mail Properties

Name	Description	Default Value
spring.mail.default-encoding	Default MimeMessage encoding.	UTF-8
spring.mail.host	SMTP server host. For instance, 'smtp.example.com'.	
spring.mail.jndi-name	Session JNDI name. When set, takes precedence over other Session settings.	

Name	Description	Default Value
spring.mail.password	Login password of the SMTP server.	
spring.mail.port	SMTP server port.	
spring.mail.properties.*	Additional JavaMail Session properties.	
spring.mail.protocol	Protocol used by the SMTP server.	smtp
spring.mail.test-connection	Whether to test that the mail server is available on startup.	false
spring.mail.username	Login user of the SMTP server.	
spring.sendgrid.api-key	SendGrid API key.	
spring.sendgrid.proxy.host	SendGrid proxy host.	
spring.sendgrid.proxy.port	SendGrid proxy port.	

.A.4. JSON Properties

Name	Description	Default Value
spring.gson.date-format	Format to use when serializing Date objects.	
spring.gson.disable-html-escaping	Whether to disable the escaping of HTML characters such as '<', '>', etc.	
spring.gson.disable-inner-class-serialization	Whether to exclude inner classes during serialization.	
spring.gson.enable-complex-map-key-serialization	Whether to enable serialization of complex map keys (i.e. non-primitives).	
spring.gson.exclude-fields-without-expose-annotation	Whether to exclude all fields from consideration for serialization or deserialization that do not have the "Expose" annotation.	
spring.gson.field-naming-policy	Naming policy that should be applied to an object's field during serialization and deserialization.	

Name	Description	Default Value
spring.gson.generate-non-executable-json	Whether to generate non-executable JSON by prefixing the output with some special text.	
spring.gson.lenient	Whether to be lenient about parsing JSON that doesn't conform to RFC 4627.	
spring.gson.long-serialization-policy	Serialization policy for Long and long types.	
spring.gson.pretty-printing	Whether to output serialized JSON that fits in a page for pretty printing.	
spring.gson.serialize-nulls	Whether to serialize null fields.	
spring.jackson.constructor-detector	Strategy to use to auto-detect constructor, and in particular behavior with single-argument constructors.	default
spring.jackson.date-format	Date format string or a fully-qualified date format class name. For instance, 'yyyy-MM-dd HH:mm:ss'.	
spring.jackson.default-leniency	Global default setting (if any) for leniency.	
spring.jackson.default-property-inclusion	Controls the inclusion of properties during serialization. Configured with one of the values in Jackson's JsonInclude.Include enumeration.	
spring.jackson.deserialization.*	Jackson on/off features that affect the way Java objects are deserialized.	
spring.jackson.generator.*	Jackson on/off features for generators.	
spring.jackson.locale	Locale used for formatting.	
spring.jackson.mapper.*	Jackson general purpose on/off features.	

Name	Description	Default Value
spring.jackson.parser.*	Jackson on/off features for parsers.	
spring.jackson.property-naming-strategy	One of the constants on Jackson's PropertyNamingStrategies. Can also be a fully-qualified class name of a PropertyNamingStrategy implementation.	
spring.jackson.serialization.*	Jackson on/off features that affect the way Java objects are serialized.	
spring.jackson.time-zone	Time zone used when formatting dates. For instance, "America/Los_Angeles" or "GMT+10".	
spring.jackson.visibility.*	Jackson visibility thresholds that can be used to limit which methods (and fields) are auto-detected.	

.A.5. Data Properties

Name	Description	Default Value
spring.cassandra.compression	Compression supported by the Cassandra binary protocol.	none
spring.cassandra.config	Location of the configuration file to use.	
spring.cassandra.connection.connect-timeout	Timeout to use when establishing driver connections.	5s
spring.cassandra.connection.init-query-timeout	Timeout to use for internal queries that run as part of the initialization process, just after a connection is opened.	5s
spring.cassandra.contact-points	Cluster node addresses in the form 'host:port', or a simple 'host' to use the configured port.	[127.0.0.1:9042]

Name	Description	Default Value
spring.cassandra.controlconnection.timeout	Timeout to use for control queries.	5s
spring.cassandra.keyspace-name	Keyspace name to use.	
spring.cassandra.local-datacenter	Datacenter that is considered "local". Contact points should be from this datacenter.	
spring.cassandra.password	Login password of the server.	
spring.cassandra.pool.heartbeat-interval	Heartbeat interval after which a message is sent on an idle connection to make sure it's still alive.	30s
spring.cassandra.pool.idle-timeout	Idle timeout before an idle connection is removed.	5s
spring.cassandra.port	Port to use if a contact point does not specify one.	9042
spring.cassandra.request.consistency	Queries consistency level.	
spring.cassandra.request.page-size	How many rows will be retrieved simultaneously in a single network round-trip.	5000
spring.cassandra.request.serial-consistency	Queries serial consistency level.	
spring.cassandra.request.throttler.drain-interval	How often the throttler attempts to dequeue requests. Set this high enough that each attempt will process multiple entries in the queue, but not delay requests too much.	
spring.cassandra.request.throttler.max-concurrent-requests	Maximum number of requests that are allowed to execute in parallel.	
spring.cassandra.request.throttler.max-queue-size	Maximum number of requests that can be enqueued when the throttling threshold is exceeded.	
spring.cassandra.request.throttler.max-requests-per-second	Maximum allowed request rate.	

Name	Description	Default Value
spring.cassandra.request.throttler.type	Request throttling type.	none
spring.cassandra.request.timeout	How long the driver waits for a request to complete.	2s
spring.cassandra.schema-action	Schema action to take at startup.	none
spring.cassandra.session-name	Name of the Cassandra session.	
spring.cassandra.ssl.bundle	SSL bundle name.	
spring.cassandra.ssl.enabled	Whether to enable SSL support.	
spring.cassandra.username	Login user of the server.	
spring.couchbase.connection-string	Connection string used to locate the Couchbase cluster.	
spring.couchbase.env.io.idle-http-connection-timeout	Length of time an HTTP connection may remain idle before it is closed and removed from the pool.	1s
spring.couchbase.env.io.max-endpoints	Maximum number of sockets per node.	12
spring.couchbase.env.io.min-endpoints	Minimum number of sockets per node.	1
spring.couchbase.env.ssl.bundle	SSL bundle name.	
spring.couchbase.env.ssl.enabled	Whether to enable SSL support. Enabled automatically if a "keyStore" or "bundle" is provided unless specified otherwise.	
spring.couchbase.env.timeouts.analytics	Timeout for the analytics service.	75s
spring.couchbase.env.timeouts.connect	Bucket connect timeout.	10s
spring.couchbase.env.timeouts.disconnect	Bucket disconnect timeout.	10s
spring.couchbase.env.timeouts.key-value	Timeout for operations on a specific key-value.	2500ms
spring.couchbase.env.timeouts.key-value-durable	Timeout for operations on a specific key-value with a durability level.	10s

Name	Description	Default Value
spring.couchbase.env.timeouts.management	Timeout for the management operations.	75s
spring.couchbase.env.timeouts.query	N1QL query operations timeout.	75s
spring.couchbase.env.timeouts.search	Timeout for the search service.	75s
spring.couchbase.env.timeouts.view	Regular and geospatial view operations timeout.	75s
spring.couchbase.password	Cluster password.	
spring.couchbase.username	Cluster username.	
spring.dao.exceptiontranslation.enabled	Whether to enable the PersistenceExceptionTranslationPostProcessor.	true
spring.data.cassandra.repositories.type	Type of Cassandra repositories to enable.	auto
spring.data.couchbase.auto-index	Automatically create views and indexes. Use the meta-data provided by "@ViewIndexed", "@N1qlPrimaryIndexed" and "@N1qlSecondaryIndexed".	false
spring.data.couchbase.bucket-name	Name of the bucket to connect to.	
spring.data.couchbase.field-naming-strategy	Fully qualified name of the FieldNamingStrategy to use.	
spring.data.couchbase.repositories.type	Type of Couchbase repositories to enable.	auto
spring.data.couchbase.scope-name	Name of the scope used for all collection access.	
spring.data.couchbase.type-key	Name of the field that stores the type information for complex types when using "MappingCouchbaseConverter".	_class
spring.data.elasticsearch.repositories.enabled	Whether to enable Elasticsearch repositories.	true
spring.data.jdbc.repositories.enabled	Whether to enable JDBC repositories.	true

Name	Description	Default Value
spring.data.jpa.repositories.bootstrap-mode	Bootstrap mode for JPA repositories.	default
spring.data.jpa.repositories.enabled	Whether to enable JPA repositories.	true
spring.data.ldap.repositories.enabled	Whether to enable LDAP repositories.	true
spring.data.mongodb.additional-hosts	Additional server hosts. Cannot be set with URI or if 'host' is not specified. Additional hosts will use the default mongo port of 27017. If you want to use a different port you can use the "host:port" syntax.	
spring.data.mongodb.authentication-database	Authentication database name.	
spring.data.mongodb.auto-index-creation	Whether to enable auto-index creation.	
spring.data.mongodb.database	Database name. Overrides database in URI.	
spring.data.mongodb.field-naming-strategy	Fully qualified name of the FieldNamingStrategy to use.	
spring.data.mongodb.gridfs.bucket	GridFS bucket name.	
spring.data.mongodb.gridfs.database	GridFS database name.	
spring.data.mongodb.host	Mongo server host. Cannot be set with URI.	
spring.data.mongodb.password	Login password of the mongo server. Cannot be set with URI.	
spring.data.mongodb.port	Mongo server port. Cannot be set with URI.	
spring.data.mongodb.replica-set-name	Required replica set name for the cluster. Cannot be set with URI.	
spring.data.mongodb.repositories.type	Type of Mongo repositories to enable.	auto
spring.data.mongodb.ssl.bundle	SSL bundle name.	

Name	Description	Default Value
spring.data.mongodb.ssl.enabled	Whether to enable SSL support. Enabled automatically if "bundle" is provided unless specified otherwise.	
spring.data.mongodb.uri	Mongo database URI. Overrides host, port, username, and password.	mongodb://localhost/test
spring.data.mongodb.username	Login user of the mongo server. Cannot be set with URI.	
spring.data.mongodb.uuid-representation	Representation to use when converting a UUID to a BSON binary value.	java-legacy
spring.data.neo4j.database	Database name to use. By default, the server decides the default database to use.	
spring.data.neo4j.repositories.type	Type of Neo4j repositories to enable.	auto
spring.data.r2dbc.repositories.enabled	Whether to enable R2DBC repositories.	true
spring.data.redis.client-name	Client name to be set on connections with CLIENT SETNAME.	
spring.data.redis.client-type	Type of client to use. By default, auto-detected according to the classpath.	
spring.data.redis.cluster.max-redirects	Maximum number of redirects to follow when executing commands across the cluster.	
spring.data.redis.cluster.nodes	Comma-separated list of "host:port" pairs to bootstrap from. This represents an "initial" list of cluster nodes and is required to have at least one entry.	
spring.data.redis.connect-timeout	Connection timeout.	
spring.data.redis.database	Database index used by the connection factory.	0

Name	Description	Default Value
spring.data.redis.host	Redis server host.	localhost
spring.data.redis.jedis.pool.enabled	Whether to enable the pool. Enabled automatically if "commons-pool2" is available. With Jedis, pooling is implicitly enabled in sentinel mode and this setting only applies to single node setup.	
spring.data.redis.jedis.pool.max-active	Maximum number of connections that can be allocated by the pool at a given time. Use a negative value for no limit.	8
spring.data.redis.jedis.pool.max-idle	Maximum number of "idle" connections in the pool. Use a negative value to indicate an unlimited number of idle connections.	8
spring.data.redis.jedis.pool.max-wait	Maximum amount of time a connection allocation should block before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely.	-1ms
spring.data.redis.jedis.pool.min-idle	Target for the minimum number of idle connections to maintain in the pool. This setting only has an effect if both it and time between eviction runs are positive.	0
spring.data.redis.jedis.pool.time-between-eviction-runs	Time between runs of the idle object evictor thread. When positive, the idle object evictor thread starts, otherwise no idle object eviction is performed.	
spring.data.redis.lettuce.cluster.refresh.adaptive	Whether adaptive topology refreshing using all available refresh triggers should be used.	false

Name	Description	Default Value
spring.data.redis.lettuce.cluster.refresh.dynamic-refresh-sources	Whether to discover and query all cluster nodes for obtaining the cluster topology. When set to false, only the initial seed nodes are used as sources for topology discovery.	true
spring.data.redis.lettuce.cluster.refresh.period	Cluster topology refresh period.	
spring.data.redis.lettuce.pool.enabled	Whether to enable the pool. Enabled automatically if "commons-pool2" is available. With Jedis, pooling is implicitly enabled in sentinel mode and this setting only applies to single node setup.	
spring.data.redis.lettuce.pool.max-active	Maximum number of connections that can be allocated by the pool at a given time. Use a negative value for no limit.	8
spring.data.redis.lettuce.pool.max-idle	Maximum number of "idle" connections in the pool. Use a negative value to indicate an unlimited number of idle connections.	8
spring.data.redis.lettuce.pool.max-wait	Maximum amount of time a connection allocation should block before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely.	-1ms
spring.data.redis.lettuce.pool.min-idle	Target for the minimum number of idle connections to maintain in the pool. This setting only has an effect if both it and time between eviction runs are positive.	0

Name	Description	Default Value
spring.data.redis.lettuce.pool.time-between-eviction-runs	Time between runs of the idle object evictor thread. When positive, the idle object evictor thread starts, otherwise no idle object eviction is performed.	
spring.data.redis.lettuce.shutdown-timeout	Shutdown timeout.	100ms
spring.data.redis.password	Login password of the redis server.	
spring.data.redis.port	Redis server port.	6379
spring.data.redis.repositories.enabled	Whether to enable Redis repositories.	true
spring.data.redis.sentinel.master	Name of the Redis server.	
spring.data.redis.sentinel.nodes	Comma-separated list of "host:port" pairs.	
spring.data.redis.sentinel.password	Password for authenticating with sentinel(s).	
spring.data.redis.sentinel.username	Login username for authenticating with sentinel(s).	
spring.data.redis.ssl.bundle	SSL bundle name.	
spring.data.redis.ssl.enabled	Whether to enable SSL support. Enabled automatically if "bundle" is provided unless specified otherwise.	
spring.data.redis.timeout	Read timeout.	
spring.data.redis.url	Connection URL. Overrides host, port, username, and password. Example: redis://user:password@example.com:6379	
spring.data.redis.username	Login username of the redis server.	
spring.data.rest.base-path	Base path to be used by Spring Data REST to expose repository resources.	

Name	Description	Default Value
spring.data.rest.default-media-type	Content type to use as a default when none is specified.	
spring.data.rest.default-page-size	Default size of pages.	
spring.data.rest.detection-strategy	Strategy to use to determine which repositories get exposed.	default
spring.data.rest.enable-enum-translation	Whether to enable enum value translation through the Spring Data REST default resource bundle.	
spring.data.rest.limit-param-name	Name of the URL query string parameter that indicates how many results to return at once.	
spring.data.rest.max-page-size	Maximum size of pages.	
spring.data.rest.page-param-name	Name of the URL query string parameter that indicates what page to return.	
spring.data.rest.return-body-on-create	Whether to return a response body after creating an entity.	
spring.data.rest.return-body-on-update	Whether to return a response body after updating an entity.	
spring.data.rest.sort-param-name	Name of the URL query string parameter that indicates what direction to sort results.	
spring.data.web.pageable.default-page-size	Default page size.	20
spring.data.web.pageable.max-page-size	Maximum page size to be accepted.	2000
spring.data.web.pageable.one-indexed-parameters	Whether to expose and assume 1-based page number indexes. Defaults to "false", meaning a page number of 0 in the request equals the first page.	false

Name	Description	Default Value
`spring.data.web.pageable.page-parameter`	Page index parameter name.	`page`
`spring.data.web.pageable.prefix`	General prefix to be prepended to the page number and page size parameters.	
`spring.data.web.pageable.qualifier-delimiter`	Delimiter to be used between the qualifier and the actual page number and size properties.	`_`
`spring.data.web.pageable.size-parameter`	Page size parameter name.	`size`
`spring.data.web.sort.sort-parameter`	Sort parameter name.	`sort`

Name	Description	Default Value
spring.datasource.dbcp2.abandoned-usage-tracking spring.datasource.dbcp2.access-to-underlying-connection-allowed spring.datasource.dbcp2.auto-commit-on-return spring.datasource.dbcp2.cache-state spring.datasource.dbcp2.clear-statement-pool-on-return spring.datasource.dbcp2.connection-factory-class-name spring.datasource.dbcp2.connection-init-sqls spring.datasource.dbcp2.default-auto-commit spring.datasource.dbcp2.default-catalog spring.datasource.dbcp2.default-query-timeout spring.datasource.dbcp2.default-read-only spring.datasource.dbcp2.default-schema spring.datasource.dbcp2.default-transaction-isolation spring.datasource.dbcp2.disconnection-sql-codes spring.datasource.dbcp2.driver spring.datasource.dbcp2.driver-class-name spring.datasource.dbcp2.eviction-policy-class-name spring.datasource.dbcp2.fast-fail-validation spring.datasource.dbcp2.initial-size spring.datasource.dbcp2.jmx-name spring.datasource.dbcp2.lifo spring.datasource.dbcp2.log-abandoned spring.datasource.dbcp2.log-expired-connections spring.datasource.dbcp2.login-timeout spring.datasource.dbcp2.max-conn-lifetime-millis spring.datasource.dbcp2.max-idle spring.datasource.dbcp2.max-open-prepared-statements spring.datasource.dbcp2.max-total spring.datasource.dbcp2.max-wait-	Commons DBCP2 specific settings bound to an instance of DBCP2's BasicDataSource	

Name	Description	Default Value
`spring.datasource.driver-class-name`	Fully qualified name of the JDBC driver. Auto-detected based on the URL by default.	
`spring.datasource.embedded-database-connection`	Connection details for an embedded database. Defaults to the most suitable embedded database that is available on the classpath.	
`spring.datasource.generate-unique-name`	Whether to generate a random datasource name.	`true`

Name	Description	Default Value
spring.datasource.hikari.allow-pool-suspension spring.datasource.hikari.auto-commit spring.datasource.hikari.catalog spring.datasource.hikari.connection-init-sql spring.datasource.hikari.connection-test-query spring.datasource.hikari.connection-timeout spring.datasource.hikari.data-source-class-name spring.datasource.hikari.data-source-j-n-d-i spring.datasource.hikari.data-source-properties spring.datasource.hikari.driver-class-name spring.datasource.hikari.exception-override-class-name spring.datasource.hikari.health-check-properties spring.datasource.hikari.idle-timeout spring.datasource.hikari.initialization-fail-timeout spring.datasource.hikari.isolate-internal-queries spring.datasource.hikari.jdbc-url spring.datasource.hikari.keepalive-time spring.datasource.hikari.leak-detection-threshold spring.datasource.hikari.login-timeout spring.datasource.hikari.max-lifetime spring.datasource.hikari.maximum-pool-size spring.datasource.hikari.metrics-tracker-factory spring.datasource.hikari.minimum-idle spring.datasource.hikari.password spring.datasource.hikari.pool-name spring.datasource.hikari.read-only spring.datasource.hikari.register-mbeans spring.datasource.hikari.scheduled-executor spring.datasource.hikari.schema	Hikari specific settings bound to an instance of Hikari's HikariDataSource	

Name	Description	Default Value
`spring.datasource.jndi-name`	JNDI location of the datasource. Class, url, username and password are ignored when set.	
`spring.datasource.name`	Datasource name to use if "generate-unique-name" is false. Defaults to "testdb" when using an embedded database, otherwise null.	

Name	Description	Default Value
spring.datasource.oracleucp.abandoned-connection-timeout spring.datasource.oracleucp.connection-factory-class-name spring.datasource.oracleucp.connection-factory-properties spring.datasource.oracleucp.connection-harvest-max-count spring.datasource.oracleucp.connection-harvest-trigger-count spring.datasource.oracleucp.connection-labeling-high-cost spring.datasource.oracleucp.connection-pool-name spring.datasource.oracleucp.connection-properties spring.datasource.oracleucp.connection-repurpose-threshold spring.datasource.oracleucp.connection-validation-timeout spring.datasource.oracleucp.connection-wait-timeout spring.datasource.oracleucp.data-source-name spring.datasource.oracleucp.database-name spring.datasource.oracleucp.description spring.datasource.oracleucp.fast-connection-failover-enabled spring.datasource.oracleucp.high-cost-connection-reuse-threshold spring.datasource.oracleucp.inactive-connection-timeout spring.datasource.oracleucp.initial-pool-size spring.datasource.oracleucp.login-timeout spring.datasource.oracleucp.max-connection-reuse-count spring.datasource.oracleucp.max-connection-reuse-time spring.datasource.oracleucp.max-connections-per-shard spring.datasource.oracleucp.max-idle-time spring.datasource.oracleucp.max-pool-size	Oracle UCP specific settings bound to an instance of Oracle UCP's PoolDataSource	

Name	Description	Default Value
`spring.datasource.password`	Login password of the database.	

Name	Description	Default Value
spring.datasource.tomcat.abandon-when-percentage-full spring.datasource.tomcat.access-to-underlying-connection-allowed spring.datasource.tomcat.alternate-username-allowed spring.datasource.tomcat.commit-on-return spring.datasource.tomcat.connection-properties spring.datasource.tomcat.data-source-j-n-d-i spring.datasource.tomcat.db-properties spring.datasource.tomcat.default-auto-commit spring.datasource.tomcat.default-catalog spring.datasource.tomcat.default-read-only spring.datasource.tomcat.default-transaction-isolation spring.datasource.tomcat.driver-class-name spring.datasource.tomcat.fair-queue spring.datasource.tomcat.ignore-exception-on-pre-load spring.datasource.tomcat.init-s-q-l spring.datasource.tomcat.initial-size spring.datasource.tomcat.jdbc-interceptors spring.datasource.tomcat.jmx-enabled spring.datasource.tomcat.log-abandoned spring.datasource.tomcat.log-validation-errors spring.datasource.tomcat.login-timeout spring.datasource.tomcat.max-active spring.datasource.tomcat.max-age spring.datasource.tomcat.max-idle spring.datasource.tomcat.max-wait spring.datasource.tomcat.min-evictable-idle-time-millis spring.datasource.tomcat.min-idle spring.datasource.tomcat.name spring.datasource.tomcat.num-tests-per-eviction-run	Tomcat datasource specific settings bound to an instance of Tomcat JDBC's DataSource	

Name	Description	Default Value
spring.datasource.type	Fully qualified name of the connection pool implementation to use. By default, it is auto-detected from the classpath.	
spring.datasource.url	JDBC URL of the database.	
spring.datasource.username	Login username of the database.	
spring.datasource.xa.data-source-class-name	XA datasource fully qualified name.	
spring.datasource.xa.properties.*	Properties to pass to the XA data source.	
spring.elasticsearch.connection-timeout	Connection timeout used when communicating with Elasticsearch.	1s
spring.elasticsearch.password	Password for authentication with Elasticsearch.	
spring.elasticsearch.path-prefix	Prefix added to the path of every request sent to Elasticsearch.	
spring.elasticsearch.restclient.sniffer.delay-after-failure	Delay of a sniff execution scheduled after a failure.	1m
spring.elasticsearch.restclient.sniffer.interval	Interval between consecutive ordinary sniff executions.	5m
spring.elasticsearch.restclient.ssl.bundle	SSL bundle name.	
spring.elasticsearch.socket-keep-alive	Whether to enable socket keep alive between client and Elasticsearch.	false
spring.elasticsearch.socket-timeout	Socket timeout used when communicating with Elasticsearch.	30s
spring.elasticsearch.uris	Comma-separated list of the Elasticsearch instances to use.	[http://localhost:9200]
spring.elasticsearch.username	Username for authentication with Elasticsearch.	
spring.h2.console.enabled	Whether to enable the console.	false

Name	Description	Default Value
spring.h2.console.path	Path at which the console is available.	/h2-console
spring.h2.console.settings.trace	Whether to enable trace output.	false
spring.h2.console.settings.web-admin-password	Password to access preferences and tools of H2 Console.	
spring.h2.console.settings.web-allow-others	Whether to enable remote access.	false
spring.influx.password	Login password.	
spring.influx.url	URL of the InfluxDB instance to which to connect.	
spring.influx.user	Login user.	
spring.jdbc.template.fetch-size	Number of rows that should be fetched from the database when more rows are needed. Use -1 to use the JDBC driver's default configuration.	-1
spring.jdbc.template.max-rows	Maximum number of rows. Use -1 to use the JDBC driver's default configuration.	-1
spring.jdbc.template.query-timeout	Query timeout. Default is to use the JDBC driver's default configuration. If a duration suffix is not specified, seconds will be used.	
spring.jooq.sql-dialect	SQL dialect to use. Auto-detected by default.	
spring.jpa.database	Target database to operate on, auto-detected by default. Can be alternatively set using the "databasePlatform" property.	

Name	Description	Default Value
spring.jpa.database-platform	Name of the target database to operate on, auto-detected by default. Can be alternatively set using the "Database" enum.	
spring.jpa.defer-datasource-initialization	Whether to defer DataSource initialization until after any EntityManagerFactory beans have been created and initialized.	false
spring.jpa.generate-ddl	Whether to initialize the schema on startup.	false
spring.jpa.hibernate.ddl-auto	DDL mode. This is actually a shortcut for the "hibernate.hbm2ddl.auto" property. Defaults to "create-drop" when using an embedded database and no schema manager was detected. Otherwise, defaults to "none".	
spring.jpa.hibernate.naming.implicit-strategy	Fully qualified name of the implicit naming strategy.	
spring.jpa.hibernate.naming.physical-strategy	Fully qualified name of the physical naming strategy.	
spring.jpa.mapping-resources	Mapping resources (equivalent to "mapping-file" entries in persistence.xml).	
spring.jpa.open-in-view	Register OpenEntityManagerInViewInterceptor. Binds a JPA EntityManager to the thread for the entire processing of the request.	true
spring.jpa.properties.*	Additional native properties to set on the JPA provider.	
spring.jpa.show-sql	Whether to enable logging of SQL statements.	false

Name	Description	Default Value
spring.ldap.anonymous-read-only	Whether read-only operations should use an anonymous environment. Disabled by default unless a username is set.	
spring.ldap.base	Base suffix from which all operations should originate.	
spring.ldap.base-environment.*	LDAP specification settings.	
spring.ldap.embedded.base-dn	List of base DNs.	
spring.ldap.embedded.credential.password	Embedded LDAP password.	
spring.ldap.embedded.credential.username	Embedded LDAP username.	
spring.ldap.embedded.ldif	Schema (LDIF) script resource reference.	classpath:schema.ldif
spring.ldap.embedded.port	Embedded LDAP port.	0
spring.ldap.embedded.validation.enabled	Whether to enable LDAP schema validation.	true
spring.ldap.embedded.validation.schema	Path to the custom schema.	
spring.ldap.password	Login password of the server.	
spring.ldap.template.ignore-name-not-found-exception	Whether NameNotFoundException should be ignored in searches through the LdapTemplate.	false
spring.ldap.template.ignore-partial-result-exception	Whether PartialResultException should be ignored in searches through the LdapTemplate.	false
spring.ldap.template.ignore-size-limit-exceeded-exception	Whether SizeLimitExceededException should be ignored in searches through the LdapTemplate.	true
spring.ldap.urls	LDAP URLs of the server.	
spring.ldap.username	Login username of the server.	

Name	Description	Default Value
spring.neo4j.authentication.kerberos-ticket	Kerberos ticket for connecting to the database. Mutual exclusive with a given username.	
spring.neo4j.authentication.password	Login password of the server.	
spring.neo4j.authentication.realm	Realm to connect to.	
spring.neo4j.authentication.username	Login user of the server.	
spring.neo4j.connection-timeout	Timeout for borrowing connections from the pool.	30s
spring.neo4j.max-transaction-retry-time	Maximum time transactions are allowed to retry.	30s
spring.neo4j.pool.connection-acquisition-timeout	Acquisition of new connections will be attempted for at most configured timeout.	60s
spring.neo4j.pool.idle-time-before-connection-test	Pooled connections that have been idle in the pool for longer than this threshold will be tested before they are used again.	
spring.neo4j.pool.log-leaked-sessions	Whether to log leaked sessions.	false
spring.neo4j.pool.max-connection-lifetime	Pooled connections older than this threshold will be closed and removed from the pool.	1h
spring.neo4j.pool.max-connection-pool-size	Maximum amount of connections in the connection pool towards a single database.	100
spring.neo4j.pool.metrics-enabled	Whether to enable metrics.	false
spring.neo4j.security.cert-file	Path to the file that holds the trusted certificates.	
spring.neo4j.security.encrypted	Whether the driver should use encrypted traffic.	false
spring.neo4j.security.hostname-verification-enabled	Whether hostname verification is required.	true
spring.neo4j.security.trust-strategy	Trust strategy to use.	trust-system-ca-signed-certificates

Name	Description	Default Value
spring.neo4j.uri	URI used by the driver.	bolt://localhost:7687
spring.r2dbc.generate-unique-name	Whether to generate a random database name. Ignore any configured name when enabled.	false
spring.r2dbc.name	Database name. Set if no name is specified in the url. Default to "testdb" when using an embedded database.	
spring.r2dbc.password	Login password of the database. Set if no password is specified in the url.	
spring.r2dbc.pool.enabled	Whether pooling is enabled. Requires r2dbc-pool.	true
spring.r2dbc.pool.initial-size	Initial connection pool size.	10
spring.r2dbc.pool.max-acquire-time	Maximum time to acquire a connection from the pool. By default, wait indefinitely.	
spring.r2dbc.pool.max-create-connection-time	Maximum time to wait to create a new connection. By default, wait indefinitely.	
spring.r2dbc.pool.max-idle-time	Maximum amount of time that a connection is allowed to sit idle in the pool.	30m
spring.r2dbc.pool.max-life-time	Maximum lifetime of a connection in the pool. By default, connections have an infinite lifetime.	
spring.r2dbc.pool.max-size	Maximal connection pool size.	10
spring.r2dbc.pool.max-validation-time	Maximum time to validate a connection from the pool. By default, wait indefinitely.	
spring.r2dbc.pool.min-idle	Minimal number of idle connections.	0
spring.r2dbc.pool.validation-depth	Validation depth.	local
spring.r2dbc.pool.validation-query	Validation query.	
spring.r2dbc.properties.*	Additional R2DBC options.	

Name	Description	Default Value
spring.r2dbc.url	R2DBC URL of the database. database name, username, password and pooling options specified in the url take precedence over individual options.	
spring.r2dbc.username	Login username of the database. Set if no username is specified in the url.	

.A.6. Transaction Properties

Name	Description	Default Value
spring.jta.atomikos.connectionfactory.borrow-connection-timeout	Timeout, in seconds, for borrowing connections from the pool.	30
spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag	Whether to ignore the transacted flag when creating session.	true
spring.jta.atomikos.connectionfactory.local-transaction-mode	Whether local transactions are desired.	false
spring.jta.atomikos.connectionfactory.maintenance-interval	Time, in seconds, between runs of the pool's maintenance thread.	60
spring.jta.atomikos.connectionfactory.max-idle-time	Time, in seconds, after which connections are cleaned up from the pool.	60
spring.jta.atomikos.connectionfactory.max-lifetime	Time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.	0
spring.jta.atomikos.connectionfactory.max-pool-size	Maximum size of the pool.	1
spring.jta.atomikos.connectionfactory.min-pool-size	Minimum size of the pool.	1
spring.jta.atomikos.connectionfactory.reap-timeout	Reap timeout, in seconds, for borrowed connections. 0 denotes no limit.	0
spring.jta.atomikos.connectionfactory.unique-resource-name	Unique name used to identify the resource during recovery.	jmsConnectionFactory

Name	Description	Default Value
spring.jta.atomikos.connectionfactory.xa-connection-factory-class-name	Vendor-specific implementation of XAConnectionFactory.	
spring.jta.atomikos.connectionfactory.xa-properties	Vendor-specific XA properties.	
spring.jta.atomikos.datasource.borrow-connection-timeout	Timeout, in seconds, for borrowing connections from the pool.	30
spring.jta.atomikos.datasource.concurrent-connection-validation	Whether to use concurrent connection validation.	true
spring.jta.atomikos.datasource.default-isolation-level	Default isolation level of connections provided by the pool.	
spring.jta.atomikos.datasource.login-timeout	Timeout, in seconds, for establishing a database connection.	0
spring.jta.atomikos.datasource.maintenance-interval	Time, in seconds, between runs of the pool's maintenance thread.	60
spring.jta.atomikos.datasource.max-idle-time	Time, in seconds, after which connections are cleaned up from the pool.	60
spring.jta.atomikos.datasource.max-lifetime	Time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.	0
spring.jta.atomikos.datasource.max-pool-size	Maximum size of the pool.	1
spring.jta.atomikos.datasource.min-pool-size	Minimum size of the pool.	1
spring.jta.atomikos.datasource.reap-timeout	Reap timeout, in seconds, for borrowed connections. 0 denotes no limit.	0
spring.jta.atomikos.datasource.test-query	SQL query or statement used to validate a connection before returning it.	
spring.jta.atomikos.datasource.unique-resource-name	Unique name used to identify the resource during recovery.	dataSource

Name	Description	Default Value
spring.jta.atomikos.datasource.xa-data-source-class-name	Vendor-specific implementation of XAConnectionFactory.	
spring.jta.atomikos.datasource.xa-properties	Vendor-specific XA properties.	
spring.jta.enabled	Whether to enable JTA support.	true
spring.transaction.default-timeout	Default transaction timeout. If a duration suffix is not specified, seconds will be used.	
spring.transaction.rollback-on-commit-failure	Whether to roll back on commit failures.	

.A.7. Data Migration Properties

Name	Description	Default Value
spring.flyway.baseline-description	Description to tag an existing schema with when applying a baseline.	<< Flyway Baseline >>
spring.flyway.baseline-on-migrate	Whether to automatically call baseline when migrating a non-empty schema.	false
spring.flyway.baseline-version	Version to tag an existing schema with when executing baseline.	1
spring.flyway.batch	Whether to batch SQL statements when executing them. Requires Flyway Teams.	
spring.flyway.cherry-pick	Migrations that Flyway should consider when migrating or undoing. When empty all available migrations are considered. Requires Flyway Teams.	
spring.flyway.clean-disabled	Whether to disable cleaning of the database.	true
spring.flyway.clean-on-validation-error	Whether to automatically call clean when a validation error occurs.	false

Name	Description	Default Value
spring.flyway.connect-retries	Maximum number of retries when attempting to connect to the database.	0
spring.flyway.connect-retries-interval	Maximum time between retries when attempting to connect to the database. If a duration suffix is not specified, seconds will be used.	120s
spring.flyway.create-schemas	Whether Flyway should attempt to create the schemas specified in the schemas property.	true
spring.flyway.default-schema	Default schema name managed by Flyway (case-sensitive).	
spring.flyway.detect-encoding	Whether to attempt to automatically detect SQL migration file encoding. Requires Flyway Teams.	
spring.flyway.driver-class-name	Fully qualified name of the JDBC driver. Auto-detected based on the URL by default.	
spring.flyway.enabled	Whether to enable flyway.	true
spring.flyway.encoding	Encoding of SQL migrations.	UTF-8
spring.flyway.error-overrides	Rules for the built-in error handling to override specific SQL states and error codes. Requires Flyway Teams.	
spring.flyway.execute-in-transaction	Whether Flyway should execute SQL within a transaction.	true
spring.flyway.fail-on-missing-locations	Whether to fail if a location of migration scripts doesn't exist.	false
spring.flyway.group	Whether to group all pending migrations together in the same transaction when applying them.	false

Name	Description	Default Value
spring.flyway.ignore-migration-patterns	Ignore migrations that match this comma-separated list of patterns when validating migrations. Requires Flyway Teams.	
spring.flyway.init-sqls	SQL statements to execute to initialize a connection immediately after obtaining it.	
spring.flyway.installed-by	Username recorded in the schema history table as having applied the migration.	
spring.flyway.jdbc-properties.*	Properties to pass to the JDBC driver. Requires Flyway Teams.	
spring.flyway.kerberos-config-file	Path of the Kerberos config file. Requires Flyway Teams.	
spring.flyway.license-key	Licence key for Flyway Teams.	
spring.flyway.locations	Locations of migrations scripts. Can contain the special "{vendor}" placeholder to use vendor-specific locations.	[classpath:db/migration]
spring.flyway.lock-retry-count	Maximum number of retries when trying to obtain a lock.	50
spring.flyway.loggers	Loggers Flyway should use.	[slf4j]
spring.flyway.mixed	Whether to allow mixing transactional and non-transactional statements within the same migration.	false
spring.flyway.oracle-kerberos-cache-file	Path of the Oracle Kerberos cache file. Requires Flyway Teams.	
spring.flyway.oracle-sqlplus	Whether to enable support for Oracle SQL*Plus commands. Requires Flyway Teams.	

Name	Description	Default Value
spring.flyway.oracle-sqlplus-warn	Whether to issue a warning rather than an error when a not-yet-supported Oracle SQL*Plus statement is encountered. Requires Flyway Teams.	
spring.flyway.oracle-wallet-location	Location of the Oracle Wallet, used to sign in to the database automatically. Requires Flyway Teams.	
spring.flyway.out-of-order	Whether to allow migrations to be run out of order.	false
spring.flyway.output-query-results	Whether Flyway should output a table with the results of queries when executing migrations. Requires Flyway Teams.	
spring.flyway.password	Login password of the database to migrate.	
spring.flyway.placeholder-prefix	Prefix of placeholders in migration scripts.	${
spring.flyway.placeholder-replacement	Perform placeholder replacement in migration scripts.	true
spring.flyway.placeholder-separator	Separator of default placeholders.	:
spring.flyway.placeholder-suffix	Suffix of placeholders in migration scripts.	}
spring.flyway.placeholders.*	Placeholders and their replacements to apply to sql migration scripts.	
spring.flyway.repeatable-sql-migration-prefix	File name prefix for repeatable SQL migrations.	R
spring.flyway.schemas	Scheme names managed by Flyway (case-sensitive).	
spring.flyway.script-placeholder-prefix	Prefix of placeholders in migration scripts.	FP__
spring.flyway.script-placeholder-suffix	Suffix of placeholders in migration scripts.	__

Name	Description	Default Value
spring.flyway.skip-default-callbacks	Whether to skip default callbacks. If true, only custom callbacks are used.	false
spring.flyway.skip-default-resolvers	Whether to skip default resolvers. If true, only custom resolvers are used.	false
spring.flyway.skip-executing-migrations	Whether Flyway should skip executing the contents of the migrations and only update the schema history table. Requires Flyway teams.	
spring.flyway.sql-migration-prefix	File name prefix for SQL migrations.	V
spring.flyway.sql-migration-separator	File name separator for SQL migrations.	__
spring.flyway.sql-migration-suffixes	File name suffix for SQL migrations.	[.sql]
spring.flyway.sql-server-kerberos-login-file	Path to the SQL Server Kerberos login file. Requires Flyway Teams.	
spring.flyway.stream	Whether to stream SQL migrations when executing them. Requires Flyway Teams.	
spring.flyway.table	Name of the schema history table that will be used by Flyway.	flyway_schema_history
spring.flyway.tablespace	Tablespace in which the schema history table is created. Ignored when using a database that does not support tablespaces. Defaults to the default tablespace of the connection used by Flyway.	
spring.flyway.target	Target version up to which migrations should be considered.	latest
spring.flyway.url	JDBC url of the database to migrate. If not set, the primary configured data source is used.	

Name	Description	Default Value
spring.flyway.user	Login user of the database to migrate.	
spring.flyway.validate-migration-naming	Whether to validate migrations and callbacks whose scripts do not obey the correct naming convention.	false
spring.flyway.validate-on-migrate	Whether to automatically call validate when performing a migration.	true
spring.liquibase.change-log	Change log configuration path.	classpath:/db/changelog/db.changelog-master.yaml
spring.liquibase.clear-checksums	Whether to clear all checksums in the current changelog, so they will be recalculated upon the next update.	false
spring.liquibase.contexts	Comma-separated list of runtime contexts to use.	
spring.liquibase.database-change-log-lock-table	Name of table to use for tracking concurrent Liquibase usage.	DATABASECHANGELOGLOCK
spring.liquibase.database-change-log-table	Name of table to use for tracking change history.	DATABASECHANGELOG
spring.liquibase.default-schema	Default database schema.	
spring.liquibase.driver-class-name	Fully qualified name of the JDBC driver. Auto-detected based on the URL by default.	
spring.liquibase.drop-first	Whether to first drop the database schema.	false
spring.liquibase.enabled	Whether to enable Liquibase support.	true
spring.liquibase.label-filter	Comma-separated list of runtime labels to use.	
spring.liquibase.liquibase-schema	Schema to use for Liquibase objects.	
spring.liquibase.liquibase-tablespace	Tablespace to use for Liquibase objects.	
spring.liquibase.parameters.*	Change log parameters.	

Name	Description	Default Value
spring.liquibase.password	Login password of the database to migrate.	
spring.liquibase.rollback-file	File to which rollback SQL is written when an update is performed.	
spring.liquibase.tag	Tag name to use when applying database changes. Can also be used with "rollbackFile" to generate a rollback script for all existing changes associated with that tag.	
spring.liquibase.test-rollback-on-update	Whether rollback should be tested before update is performed.	false
spring.liquibase.url	JDBC URL of the database to migrate. If not set, the primary configured data source is used.	
spring.liquibase.user	Login user of the database to migrate.	
spring.sql.init.continue-on-error	Whether initialization should continue when an error occurs.	false
spring.sql.init.data-locations	Locations of the data (DML) scripts to apply to the database.	
spring.sql.init.encoding	Encoding of the schema and data scripts.	
spring.sql.init.mode	Mode to apply when determining whether initialization should be performed.	embedded
spring.sql.init.password	Password of the database to use when applying initialization scripts (if different).	
spring.sql.init.platform	Platform to use in the default schema or data script locations, schema-${platform}.sql and data-${platform}.sql.	all

Name	Description	Default Value
`spring.sql.init.schema-locations`	Locations of the schema (DDL) scripts to apply to the database.	
`spring.sql.init.separator`	Statement separator in the schema and data scripts.	`;`
`spring.sql.init.username`	Username of the database to use when applying initialization scripts (if different).	

.A.8. Integration Properties

Name	Description	Default Value
`spring.activemq.broker-url`	URL of the ActiveMQ broker. Auto-generated by default.	
`spring.activemq.close-timeout`	Time to wait before considering a close complete.	`15s`
`spring.activemq.non-blocking-redelivery`	Whether to stop message delivery before re-delivering messages from a rolled back transaction. This implies that message order is not preserved when this is enabled.	`false`
`spring.activemq.packages.trust-all`	Whether to trust all packages.	
`spring.activemq.packages.trusted`	Comma-separated list of specific packages to trust (when not trusting all packages).	
`spring.activemq.password`	Login password of the broker.	
`spring.activemq.pool.block-if-full`	Whether to block when a connection is requested and the pool is full. Set it to false to throw a "JMSException" instead.	`true`
`spring.activemq.pool.block-if-full-timeout`	Blocking period before throwing an exception if the pool is still full.	`-1ms`

Name	Description	Default Value
spring.activemq.pool.enabled	Whether a JmsPoolConnectionFactory should be created, instead of a regular ConnectionFactory.	false
spring.activemq.pool.idle-timeout	Connection idle timeout.	30s
spring.activemq.pool.max-connections	Maximum number of pooled connections.	1
spring.activemq.pool.max-sessions-per-connection	Maximum number of pooled sessions per connection in the pool.	500
spring.activemq.pool.time-between-expiration-check	Time to sleep between runs of the idle connection eviction thread. When negative, no idle connection eviction thread runs.	-1ms
spring.activemq.pool.use-anonymous-producers	Whether to use only one anonymous "MessageProducer" instance. Set it to false to create one "MessageProducer" every time one is required.	true
spring.activemq.send-timeout	Time to wait on message sends for a response. Set it to 0 to wait forever.	0ms
spring.activemq.user	Login user of the broker.	
spring.artemis.broker-url	Artemis broker url.	tcp://localhost:61616
spring.artemis.embedded.cluster-password	Cluster password. Randomly generated on startup by default.	
spring.artemis.embedded.data-directory	Journal file directory. Not necessary if persistence is turned off.	
spring.artemis.embedded.enabled	Whether to enable embedded mode if the Artemis server APIs are available.	true
spring.artemis.embedded.persistent	Whether to enable persistent store.	false
spring.artemis.embedded.queues	Comma-separated list of queues to create on startup.	[]

Name	Description	Default Value
spring.artemis.embedded.server-id	Server ID. By default, an auto-incremented counter is used.	0
spring.artemis.embedded.topics	Comma-separated list of topics to create on startup.	[]
spring.artemis.mode	Artemis deployment mode, auto-detected by default.	
spring.artemis.password	Login password of the broker.	
spring.artemis.pool.block-if-full	Whether to block when a connection is requested and the pool is full. Set it to false to throw a "JMSException" instead.	true
spring.artemis.pool.block-if-full-timeout	Blocking period before throwing an exception if the pool is still full.	-1ms
spring.artemis.pool.enabled	Whether a JmsPoolConnectionFactory should be created, instead of a regular ConnectionFactory.	false
spring.artemis.pool.idle-timeout	Connection idle timeout.	30s
spring.artemis.pool.max-connections	Maximum number of pooled connections.	1
spring.artemis.pool.max-sessions-per-connection	Maximum number of pooled sessions per connection in the pool.	500
spring.artemis.pool.time-between-expiration-check	Time to sleep between runs of the idle connection eviction thread. When negative, no idle connection eviction thread runs.	-1ms
spring.artemis.pool.use-anonymous-producers	Whether to use only one anonymous "MessageProducer" instance. Set it to false to create one "MessageProducer" every time one is required.	true
spring.artemis.user	Login user of the broker.	
spring.batch.jdbc.initialize-schema	Database schema initialization mode.	embedded

Name	Description	Default Value
spring.batch.jdbc.isolation-level-for-create	Transaction isolation level to use when creating job meta-data for new jobs.	
spring.batch.jdbc.platform	Platform to use in initialization scripts if the @@platform@@ placeholder is used. Auto-detected by default.	
spring.batch.jdbc.schema	Path to the SQL file to use to initialize the database schema.	classpath:org/springframework/batch/core/schema-@@platform@@.sql
spring.batch.jdbc.table-prefix	Table prefix for all the batch meta-data tables.	
spring.batch.job.enabled	Execute all Spring Batch jobs in the context on startup.	true
spring.batch.job.name	Job name to execute on startup. Must be specified if multiple Jobs are found in the context.	
spring.hazelcast.config	The location of the configuration file to use to initialize Hazelcast.	
spring.integration.channel.auto-create	Whether to create input channels if necessary.	true
spring.integration.channel.max-broadcast-subscribers	Default number of subscribers allowed on, for example, a 'PublishSubscribeChannel'.	
spring.integration.channel.max-unicast-subscribers	Default number of subscribers allowed on, for example, a 'DirectChannel'.	
spring.integration.endpoint.no-auto-startup	A comma-separated list of endpoint bean names patterns that should not be started automatically during application startup.	
spring.integration.endpoint.read-only-headers	A comma-separated list of message header names that should not be populated into Message instances during a header copying operation.	

Name	Description	Default Value
spring.integration.endpoint.throw-exception-on-late-reply	Whether to throw an exception when a reply is not expected anymore by a gateway.	false
spring.integration.error.ignore-failures	Whether to ignore failures for one or more of the handlers of the global 'errorChannel'.	true
spring.integration.error.require-subscribers	Whether to not silently ignore messages on the global 'errorChannel' when there are no subscribers.	true
spring.integration.jdbc.initialize-schema	Database schema initialization mode.	embedded
spring.integration.jdbc.platform	Platform to use in initialization scripts if the @@platform@@ placeholder is used. Auto-detected by default.	
spring.integration.jdbc.schema	Path to the SQL file to use to initialize the database schema.	classpath:org/springframework/integration/jdbc/schema-@@platform@@.sql
spring.integration.management.default-logging-enabled	Whether Spring Integration components should perform logging in the main message flow. When disabled, such logging will be skipped without checking the logging level. When enabled, such logging is controlled as normal by the logging system's log level configuration.	true

Name	Description	Default Value
spring.integration.management.observation-patterns	Comma-separated list of simple patterns to match against the names of Spring Integration components. When matched, observation instrumentation will be performed for the component. Please refer to the javadoc of the smartMatch method of Spring Integration's PatternMatchUtils for details of the pattern syntax.	
spring.integration.poller.cron	Cron expression for polling. Mutually exclusive with 'fixedDelay' and 'fixedRate'.	
spring.integration.poller.fixed-delay	Polling delay period. Mutually exclusive with 'cron' and 'fixedRate'.	
spring.integration.poller.fixed-rate	Polling rate period. Mutually exclusive with 'fixedDelay' and 'cron'.	
spring.integration.poller.initial-delay	Polling initial delay. Applied for 'fixedDelay' and 'fixedRate'; ignored for 'cron'.	
spring.integration.poller.max-messages-per-poll	Maximum number of messages to poll per polling cycle.	
spring.integration.poller.receive-timeout	How long to wait for messages on poll.	1s
spring.integration.rsocket.client.host	TCP RSocket server host to connect to.	
spring.integration.rsocket.client.port	TCP RSocket server port to connect to.	
spring.integration.rsocket.client.uri	WebSocket RSocket server uri to connect to.	
spring.integration.rsocket.server.message-mapping-enabled	Whether to handle message mapping for RSocket through Spring Integration.	false
spring.jms.cache.consumers	Whether to cache message consumers.	false

Name	Description	Default Value
`spring.jms.cache.enabled`	Whether to cache sessions.	`true`
`spring.jms.cache.producers`	Whether to cache message producers.	`true`
`spring.jms.cache.session-cache-size`	Size of the session cache (per JMS Session type).	`1`
`spring.jms.jndi-name`	Connection factory JNDI name. When set, takes precedence to others connection factory auto-configurations.	
`spring.jms.listener.acknowledge-mode`	Acknowledge mode of the container. By default, the listener is transacted with automatic acknowledgment.	
`spring.jms.listener.auto-startup`	Start the container automatically on startup.	`true`
`spring.jms.listener.concurrency`	Minimum number of concurrent consumers. When max-concurrency is not specified the minimum will also be used as the maximum.	
`spring.jms.listener.max-concurrency`	Maximum number of concurrent consumers.	
`spring.jms.listener.receive-timeout`	Timeout to use for receive calls. Use -1 for a no-wait receive or 0 for no timeout at all. The latter is only feasible if not running within a transaction manager and is generally discouraged since it prevents clean shutdown.	`1s`
`spring.jms.pub-sub-domain`	Whether the default destination type is topic.	`false`
`spring.jms.template.default-destination`	Default destination to use on send and receive operations that do not have a destination parameter.	
`spring.jms.template.delivery-delay`	Delivery delay to use for send calls.	

Name	Description	Default Value
spring.jms.template.delivery-mode	Delivery mode. Enables QoS (Quality of Service) when set.	
spring.jms.template.priority	Priority of a message when sending. Enables QoS (Quality of Service) when set.	
spring.jms.template.qos-enabled	Whether to enable explicit QoS (Quality of Service) when sending a message. When enabled, the delivery mode, priority and time-to-live properties will be used when sending a message. QoS is automatically enabled when at least one of those settings is customized.	
spring.jms.template.receive-timeout	Timeout to use for receive calls.	
spring.jms.template.time-to-live	Time-to-live of a message when sending. Enables QoS (Quality of Service) when set.	
spring.kafka.admin.auto-create	Whether to automatically create topics during context initialization. When set to false, disables automatic topic creation during context initialization.	true
spring.kafka.admin.client-id	ID to pass to the server when making requests. Used for server-side logging.	
spring.kafka.admin.close-timeout	Close timeout.	
spring.kafka.admin.fail-fast	Whether to fail fast if the broker is not available on startup.	false
spring.kafka.admin.modify-topic-configs	Whether to enable modification of existing topic configuration.	false
spring.kafka.admin.operation-timeout	Operation timeout.	

Name	Description	Default Value
spring.kafka.admin.properties.*	Additional admin-specific properties used to configure the client.	
spring.kafka.admin.security.protocol	Security protocol used to communicate with brokers.	
spring.kafka.admin.ssl.key-password	Password of the private key in either key store key or key store file.	
spring.kafka.admin.ssl.key-store-certificate-chain	Certificate chain in PEM format with a list of X.509 certificates.	
spring.kafka.admin.ssl.key-store-key	Private key in PEM format with PKCS#8 keys.	
spring.kafka.admin.ssl.key-store-location	Location of the key store file.	
spring.kafka.admin.ssl.key-store-password	Store password for the key store file.	
spring.kafka.admin.ssl.key-store-type	Type of the key store.	
spring.kafka.admin.ssl.protocol	SSL protocol to use.	
spring.kafka.admin.ssl.trust-store-certificates	Trusted certificates in PEM format with X.509 certificates.	
spring.kafka.admin.ssl.trust-store-location	Location of the trust store file.	
spring.kafka.admin.ssl.trust-store-password	Store password for the trust store file.	
spring.kafka.admin.ssl.trust-store-type	Type of the trust store.	
spring.kafka.bootstrap-servers	Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Applies to all components unless overridden.	
spring.kafka.client-id	ID to pass to the server when making requests. Used for server-side logging.	

Name	Description	Default Value
spring.kafka.consumer.auto-commit-interval	Frequency with which the consumer offsets are auto-committed to Kafka if 'enable.auto.commit' is set to true.	
spring.kafka.consumer.auto-offset-reset	What to do when there is no initial offset in Kafka or if the current offset no longer exists on the server.	
spring.kafka.consumer.bootstrap-servers	Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Overrides the global property, for consumers.	
spring.kafka.consumer.client-id	ID to pass to the server when making requests. Used for server-side logging.	
spring.kafka.consumer.enable-auto-commit	Whether the consumer's offset is periodically committed in the background.	
spring.kafka.consumer.fetch-max-wait	Maximum amount of time the server blocks before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by "fetch-min-size".	
spring.kafka.consumer.fetch-min-size	Minimum amount of data the server should return for a fetch request.	
spring.kafka.consumer.group-id	Unique string that identifies the consumer group to which this consumer belongs.	
spring.kafka.consumer.heartbeat-interval	Expected time between heartbeats to the consumer coordinator.	
spring.kafka.consumer.isolation-level	Isolation level for reading messages that have been written transactionally.	read-uncommitted

Name	Description	Default Value
spring.kafka.consumer.key-deserializer	Deserializer class for keys.	
spring.kafka.consumer.max-poll-records	Maximum number of records returned in a single call to poll().	
spring.kafka.consumer.properties.*	Additional consumer-specific properties used to configure the client.	
spring.kafka.consumer.security.protocol	Security protocol used to communicate with brokers.	
spring.kafka.consumer.ssl.key-password	Password of the private key in either key store key or key store file.	
spring.kafka.consumer.ssl.key-store-certificate-chain	Certificate chain in PEM format with a list of X.509 certificates.	
spring.kafka.consumer.ssl.key-store-key	Private key in PEM format with PKCS#8 keys.	
spring.kafka.consumer.ssl.key-store-location	Location of the key store file.	
spring.kafka.consumer.ssl.key-store-password	Store password for the key store file.	
spring.kafka.consumer.ssl.key-store-type	Type of the key store.	
spring.kafka.consumer.ssl.protocol	SSL protocol to use.	
spring.kafka.consumer.ssl.trust-store-certificates	Trusted certificates in PEM format with X.509 certificates.	
spring.kafka.consumer.ssl.trust-store-location	Location of the trust store file.	
spring.kafka.consumer.ssl.trust-store-password	Store password for the trust store file.	
spring.kafka.consumer.ssl.trust-store-type	Type of the trust store.	
spring.kafka.consumer.value-deserializer	Deserializer class for values.	
spring.kafka.jaas.control-flag	Control flag for login configuration.	required
spring.kafka.jaas.enabled	Whether to enable JAAS configuration.	false

Name	Description	Default Value
spring.kafka.jaas.login-module	Login module.	com.sun.security.auth.module.Krb5LoginModule
spring.kafka.jaas.options.*	Additional JAAS options.	
spring.kafka.listener.ack-count	Number of records between offset commits when ackMode is "COUNT" or "COUNT_TIME".	
spring.kafka.listener.ack-mode	Listener AckMode. See the spring-kafka documentation.	
spring.kafka.listener.ack-time	Time between offset commits when ackMode is "TIME" or "COUNT_TIME".	
spring.kafka.listener.async-acks	Support for asynchronous record acknowledgements. Only applies when spring.kafka.listener.ack-mode is manual or manual-immediate.	
spring.kafka.listener.auto-startup	Whether to auto start the container.	true
spring.kafka.listener.client-id	Prefix for the listener's consumer client.id property.	
spring.kafka.listener.concurrency	Number of threads to run in the listener containers.	
spring.kafka.listener.idle-between-polls	Sleep interval between Consumer.poll(Duration) calls.	0
spring.kafka.listener.idle-event-interval	Time between publishing idle consumer events (no data received).	
spring.kafka.listener.idle-partition-event-interval	Time between publishing idle partition consumer events (no data received for partition).	
spring.kafka.listener.immediate-stop	Whether the container stops after the current record is processed or after all the records from the previous poll are processed.	false

Name	Description	Default Value
spring.kafka.listener.log-container-config	Whether to log the container configuration during initialization (INFO level).	
spring.kafka.listener.missing-topics-fatal	Whether the container should fail to start if at least one of the configured topics are not present on the broker.	false
spring.kafka.listener.monitor-interval	Time between checks for non-responsive consumers. If a duration suffix is not specified, seconds will be used.	
spring.kafka.listener.no-poll-threshold	Multiplier applied to "pollTimeout" to determine if a consumer is non-responsive.	
spring.kafka.listener.poll-timeout	Timeout to use when polling the consumer.	
spring.kafka.listener.type	Listener type.	single
spring.kafka.producer.acks	Number of acknowledgments the producer requires the leader to have received before considering a request complete.	
spring.kafka.producer.batch-size	Default batch size. A small batch size will make batching less common and may reduce throughput (a batch size of zero disables batching entirely).	
spring.kafka.producer.bootstrap-servers	Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Overrides the global property, for producers.	
spring.kafka.producer.buffer-memory	Total memory size the producer can use to buffer records waiting to be sent to the server.	

Name	Description	Default Value
spring.kafka.producer.client-id	ID to pass to the server when making requests. Used for server-side logging.	
spring.kafka.producer.compression-type	Compression type for all data generated by the producer.	
spring.kafka.producer.key-serializer	Serializer class for keys.	
spring.kafka.producer.properties.*	Additional producer-specific properties used to configure the client.	
spring.kafka.producer.retries	When greater than zero, enables retrying of failed sends.	
spring.kafka.producer.security.protocol	Security protocol used to communicate with brokers.	
spring.kafka.producer.ssl.key-password	Password of the private key in either key store key or key store file.	
spring.kafka.producer.ssl.key-store-certificate-chain	Certificate chain in PEM format with a list of X.509 certificates.	
spring.kafka.producer.ssl.key-store-key	Private key in PEM format with PKCS#8 keys.	
spring.kafka.producer.ssl.key-store-location	Location of the key store file.	
spring.kafka.producer.ssl.key-store-password	Store password for the key store file.	
spring.kafka.producer.ssl.key-store-type	Type of the key store.	
spring.kafka.producer.ssl.protocol	SSL protocol to use.	
spring.kafka.producer.ssl.trust-store-certificates	Trusted certificates in PEM format with X.509 certificates.	
spring.kafka.producer.ssl.trust-store-location	Location of the trust store file.	
spring.kafka.producer.ssl.trust-store-password	Store password for the trust store file.	
spring.kafka.producer.ssl.trust-store-type	Type of the trust store.	

Name	Description	Default Value
spring.kafka.producer.transaction-id-prefix	When non empty, enables transaction support for producer.	
spring.kafka.producer.value-serializer	Serializer class for values.	
spring.kafka.properties.*	Additional properties, common to producers and consumers, used to configure the client.	
spring.kafka.retry.topic.attempts	Total number of processing attempts made before sending the message to the DLT.	3
spring.kafka.retry.topic.delay	Canonical backoff period. Used as an initial value in the exponential case, and as a minimum value in the uniform case.	1s
spring.kafka.retry.topic.enabled	Whether to enable topic-based non-blocking retries.	false
spring.kafka.retry.topic.max-delay	Maximum wait between retries. If less than the delay then the default of 30 seconds is applied.	0
spring.kafka.retry.topic.multiplier	Multiplier to use for generating the next backoff delay.	0
spring.kafka.retry.topic.random-back-off	Whether to have the backoff delays.	false
spring.kafka.security.protocol	Security protocol used to communicate with brokers.	
spring.kafka.ssl.key-password	Password of the private key in either key store key or key store file.	
spring.kafka.ssl.key-store-certificate-chain	Certificate chain in PEM format with a list of X.509 certificates.	
spring.kafka.ssl.key-store-key	Private key in PEM format with PKCS#8 keys.	
spring.kafka.ssl.key-store-location	Location of the key store file.	

Name	Description	Default Value
spring.kafka.ssl.key-store-password	Store password for the key store file.	
spring.kafka.ssl.key-store-type	Type of the key store.	
spring.kafka.ssl.protocol	SSL protocol to use.	
spring.kafka.ssl.trust-store-certificates	Trusted certificates in PEM format with X.509 certificates.	
spring.kafka.ssl.trust-store-location	Location of the trust store file.	
spring.kafka.ssl.trust-store-password	Store password for the trust store file.	
spring.kafka.ssl.trust-store-type	Type of the trust store.	
spring.kafka.streams.application-id	Kafka streams application.id property; default spring.application.name.	
spring.kafka.streams.auto-startup	Whether to auto-start the streams factory bean.	true
spring.kafka.streams.bootstrap-servers	Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Overrides the global property, for streams.	
spring.kafka.streams.cleanup.on-shutdown	Cleanup the application's local state directory on shutdown.	false
spring.kafka.streams.cleanup.on-startup	Cleanup the application's local state directory on startup.	false
spring.kafka.streams.client-id	ID to pass to the server when making requests. Used for server-side logging.	
spring.kafka.streams.properties.*	Additional Kafka properties used to configure the streams.	
spring.kafka.streams.replication-factor	The replication factor for change log topics and repartition topics created by the stream processing application.	

Name	Description	Default Value
spring.kafka.streams.security.protocol	Security protocol used to communicate with brokers.	
spring.kafka.streams.ssl.key-password	Password of the private key in either key store key or key store file.	
spring.kafka.streams.ssl.key-store-certificate-chain	Certificate chain in PEM format with a list of X.509 certificates.	
spring.kafka.streams.ssl.key-store-key	Private key in PEM format with PKCS#8 keys.	
spring.kafka.streams.ssl.key-store-location	Location of the key store file.	
spring.kafka.streams.ssl.key-store-password	Store password for the key store file.	
spring.kafka.streams.ssl.key-store-type	Type of the key store.	
spring.kafka.streams.ssl.protocol	SSL protocol to use.	
spring.kafka.streams.ssl.trust-store-certificates	Trusted certificates in PEM format with X.509 certificates.	
spring.kafka.streams.ssl.trust-store-location	Location of the trust store file.	
spring.kafka.streams.ssl.trust-store-password	Store password for the trust store file.	
spring.kafka.streams.ssl.trust-store-type	Type of the trust store.	
spring.kafka.streams.state-dir	Directory location for the state store.	
spring.kafka.streams.state-store-cache-max-size	Maximum size of the in-memory state store cache across all threads.	
spring.kafka.template.default-topic	Default topic to which messages are sent.	
spring.kafka.template.transaction-id-prefix	Transaction id prefix, override the transaction id prefix in the producer factory.	
spring.rabbitmq.address-shuffle-mode	Mode used to shuffle configured addresses.	none

Name	Description	Default Value
spring.rabbitmq.addresses	Comma-separated list of addresses to which the client should connect. When set, the host and port are ignored.	
spring.rabbitmq.cache.channel.checkout-timeout	Duration to wait to obtain a channel if the cache size has been reached. If 0, always create a new channel.	
spring.rabbitmq.cache.channel.size	Number of channels to retain in the cache. When "check-timeout" > 0, max channels per connection.	
spring.rabbitmq.cache.connection.mode	Connection factory cache mode.	channel
spring.rabbitmq.cache.connection.size	Number of connections to cache. Only applies when mode is CONNECTION.	
spring.rabbitmq.channel-rpc-timeout	Continuation timeout for RPC calls in channels. Set it to zero to wait forever.	10m
spring.rabbitmq.connection-timeout	Connection timeout. Set it to zero to wait forever.	
spring.rabbitmq.dynamic	Whether to create an AmqpAdmin bean.	true
spring.rabbitmq.host	RabbitMQ host. Ignored if an address is set.	localhost
spring.rabbitmq.listener.direct.acknowledge-mode	Acknowledge mode of container.	
spring.rabbitmq.listener.direct.auto-startup	Whether to start the container automatically on startup.	true
spring.rabbitmq.listener.direct.consumers-per-queue	Number of consumers per queue.	
spring.rabbitmq.listener.direct.de-batching-enabled	Whether the container should present batched messages as discrete messages or call the listener with the batch.	true
spring.rabbitmq.listener.direct.default-requeue-rejected	Whether rejected deliveries are re-queued by default.	

Name	Description	Default Value
spring.rabbitmq.listener.direct.idle-event-interval	How often idle container events should be published.	
spring.rabbitmq.listener.direct.missing-queues-fatal	Whether to fail if the queues declared by the container are not available on the broker.	false
spring.rabbitmq.listener.direct.prefetch	Maximum number of unacknowledged messages that can be outstanding at each consumer.	
spring.rabbitmq.listener.direct.retry.enabled	Whether publishing retries are enabled.	false
spring.rabbitmq.listener.direct.retry.initial-interval	Duration between the first and second attempt to deliver a message.	1000ms
spring.rabbitmq.listener.direct.retry.max-attempts	Maximum number of attempts to deliver a message.	3
spring.rabbitmq.listener.direct.retry.max-interval	Maximum duration between attempts.	10000ms
spring.rabbitmq.listener.direct.retry.multiplier	Multiplier to apply to the previous retry interval.	1
spring.rabbitmq.listener.direct.retry.stateless	Whether retries are stateless or stateful.	true
spring.rabbitmq.listener.simple.acknowledge-mode	Acknowledge mode of container.	
spring.rabbitmq.listener.simple.auto-startup	Whether to start the container automatically on startup.	true
spring.rabbitmq.listener.simple.batch-size	Batch size, expressed as the number of physical messages, to be used by the container.	
spring.rabbitmq.listener.simple.concurrency	Minimum number of listener invoker threads.	

Name	Description	Default Value
spring.rabbitmq.listener.simple.consumer-batch-enabled	Whether the container creates a batch of messages based on the 'receive-timeout' and 'batch-size'. Coerces 'de-batching-enabled' to true to include the contents of a producer created batch in the batch as discrete records.	false
spring.rabbitmq.listener.simple.de-batching-enabled	Whether the container should present batched messages as discrete messages or call the listener with the batch.	true
spring.rabbitmq.listener.simple.default-requeue-rejected	Whether rejected deliveries are re-queued by default.	
spring.rabbitmq.listener.simple.idle-event-interval	How often idle container events should be published.	
spring.rabbitmq.listener.simple.max-concurrency	Maximum number of listener invoker threads.	
spring.rabbitmq.listener.simple.missing-queues-fatal	Whether to fail if the queues declared by the container are not available on the broker and/or whether to stop the container if one or more queues are deleted at runtime.	true
spring.rabbitmq.listener.simple.prefetch	Maximum number of unacknowledged messages that can be outstanding at each consumer.	
spring.rabbitmq.listener.simple.retry.enabled	Whether publishing retries are enabled.	false
spring.rabbitmq.listener.simple.retry.initial-interval	Duration between the first and second attempt to deliver a message.	1000ms
spring.rabbitmq.listener.simple.retry.max-attempts	Maximum number of attempts to deliver a message.	3
spring.rabbitmq.listener.simple.retry.max-interval	Maximum duration between attempts.	10000ms

Name	Description	Default Value
spring.rabbitmq.listener.simple.retry.multiplier	Multiplier to apply to the previous retry interval.	1
spring.rabbitmq.listener.simple.retry.stateless	Whether retries are stateless or stateful.	true
spring.rabbitmq.listener.stream.auto-startup	Whether to start the container automatically on startup.	true
spring.rabbitmq.listener.stream.native-listener	Whether the container will support listeners that consume native stream messages instead of Spring AMQP messages.	false
spring.rabbitmq.listener.type	Listener container type.	simple
spring.rabbitmq.password	Login to authenticate against the broker.	guest
spring.rabbitmq.port	RabbitMQ port. Ignored if an address is set. Default to 5672, or 5671 if SSL is enabled.	
spring.rabbitmq.publisher-confirm-type	Type of publisher confirms to use.	
spring.rabbitmq.publisher-returns	Whether to enable publisher returns.	false
spring.rabbitmq.requested-channel-max	Number of channels per connection requested by the client. Use 0 for unlimited.	2047
spring.rabbitmq.requested-heartbeat	Requested heartbeat timeout; zero for none. If a duration suffix is not specified, seconds will be used.	
spring.rabbitmq.ssl.algorithm	SSL algorithm to use. By default, configured by the Rabbit client library.	
spring.rabbitmq.ssl.enabled	Whether to enable SSL support. Determined automatically if an address is provided with the protocol (amqp:// vs. amqps://).	

Name	Description	Default Value
spring.rabbitmq.ssl.key-store	Path to the key store that holds the SSL certificate.	
spring.rabbitmq.ssl.key-store-algorithm	Key store algorithm.	SunX509
spring.rabbitmq.ssl.key-store-password	Password used to access the key store.	
spring.rabbitmq.ssl.key-store-type	Key store type.	PKCS12
spring.rabbitmq.ssl.trust-store	Trust store that holds SSL certificates.	
spring.rabbitmq.ssl.trust-store-algorithm	Trust store algorithm.	SunX509
spring.rabbitmq.ssl.trust-store-password	Password used to access the trust store.	
spring.rabbitmq.ssl.trust-store-type	Trust store type.	JKS
spring.rabbitmq.ssl.validate-server-certificate	Whether to enable server side certificate validation.	true
spring.rabbitmq.ssl.verify-hostname	Whether to enable hostname verification.	true
spring.rabbitmq.stream.host	Host of a RabbitMQ instance with the Stream plugin enabled.	localhost
spring.rabbitmq.stream.name	Name of the stream.	
spring.rabbitmq.stream.password	Login password to authenticate to the broker. When not set spring.rabbitmq.password is used.	
spring.rabbitmq.stream.port	Stream port of a RabbitMQ instance with the Stream plugin enabled.	
spring.rabbitmq.stream.username	Login user to authenticate to the broker. When not set, spring.rabbitmq.username is used.	
spring.rabbitmq.template.default-receive-queue	Name of the default queue to receive messages from when none is specified explicitly.	

Name	Description	Default Value
spring.rabbitmq.template.exchange	Name of the default exchange to use for send operations.	
spring.rabbitmq.template.mandatory	Whether to enable mandatory messages.	
spring.rabbitmq.template.receive-timeout	Timeout for receive() operations.	
spring.rabbitmq.template.reply-timeout	Timeout for sendAndReceive() operations.	
spring.rabbitmq.template.retry.enabled	Whether publishing retries are enabled.	false
spring.rabbitmq.template.retry.initial-interval	Duration between the first and second attempt to deliver a message.	1000ms
spring.rabbitmq.template.retry.max-attempts	Maximum number of attempts to deliver a message.	3
spring.rabbitmq.template.retry.max-interval	Maximum duration between attempts.	10000ms
spring.rabbitmq.template.retry.multiplier	Multiplier to apply to the previous retry interval.	1
spring.rabbitmq.template.routing-key	Value of a default routing key to use for send operations.	
spring.rabbitmq.username	Login user to authenticate to the broker.	guest
spring.rabbitmq.virtual-host	Virtual host to use when connecting to the broker.	
spring.webservices.path	Path that serves as the base URI for the services.	/services
spring.webservices.servlet.init.*	Servlet init parameters to pass to Spring Web Services.	
spring.webservices.servlet.load-on-startup	Load on startup priority of the Spring Web Services servlet.	-1
spring.webservices.wsdl-locations	Comma-separated list of locations of WSDLs and accompanying XSDs to be exposed as beans.	

.A.9. Web Properties

Name	Description	Default Value
spring.graphql.cors.allow-credentials	Whether credentials are supported. When not set, credentials are not supported.	
spring.graphql.cors.allowed-headers	Comma-separated list of HTTP headers to allow in a request. '*' allows all headers.	
spring.graphql.cors.allowed-methods	Comma-separated list of HTTP methods to allow. '*' allows all methods. When not set, defaults to GET.	
spring.graphql.cors.allowed-origin-patterns	Comma-separated list of origin patterns to allow. Unlike allowed origins which only support '*', origin patterns are more flexible, e.g. 'https://*.example.com', and can be used with allow-credentials. When neither allowed origins nor allowed origin patterns are set, cross-origin requests are effectively disabled.	
spring.graphql.cors.allowed-origins	Comma-separated list of origins to allow with '*' allowing all origins. When allow-credentials is enabled, '*' cannot be used, and setting origin patterns should be considered instead. When neither allowed origins nor allowed origin patterns are set, cross-origin requests are effectively disabled.	
spring.graphql.cors.exposed-headers	Comma-separated list of headers to include in a response.	

Name	Description	Default Value
spring.graphql.cors.max-age	How long the response from a pre-flight request can be cached by clients. If a duration suffix is not specified, seconds will be used.	1800s
spring.graphql.graphiql.enabled	Whether the default GraphiQL UI is enabled.	false
spring.graphql.graphiql.path	Path to the GraphiQL UI endpoint.	/graphiql
spring.graphql.path	Path at which to expose a GraphQL request HTTP endpoint.	/graphql
spring.graphql.rsocket.mapping	Mapping of the RSocket message handler.	
spring.graphql.schema.file-extensions	File extensions for GraphQL schema files.	.graphqls,.gqls
spring.graphql.schema.introspection.enabled	Whether field introspection should be enabled at the schema level.	true
spring.graphql.schema.locations	Locations of GraphQL schema files.	classpath:graphql/**/
spring.graphql.schema.printer.enabled	Whether the endpoint that prints the schema is enabled. Schema is available under spring.graphql.path + "/schema".	false
spring.graphql.websocket.connection-init-timeout	Time within which the initial {@code CONNECTION_INIT} type message must be received.	60s
spring.graphql.websocket.path	Path of the GraphQL WebSocket subscription endpoint.	
spring.hateoas.use-hal-as-default-json-media-type	Whether application/hal+json responses should be sent to requests that accept application/json.	true

Name	Description	Default Value
spring.jersey.application-path	Path that serves as the base URI for the application. If specified, overrides the value of "@ApplicationPath".	
spring.jersey.filter.order	Jersey filter chain order.	0
spring.jersey.init.*	Init parameters to pass to Jersey through the servlet or filter.	
spring.jersey.servlet.load-on-startup	Load on startup priority of the Jersey servlet.	-1
spring.jersey.type	Jersey integration type.	servlet
spring.mvc.async.request-timeout	Amount of time before asynchronous request handling times out. If this value is not set, the default timeout of the underlying implementation is used.	
spring.mvc.contentnegotiation.favor-parameter	Whether a request parameter ("format" by default) should be used to determine the requested media type.	false
spring.mvc.contentnegotiation.media-types.*	Map file extensions to media types for content negotiation. For instance, yml to text/yaml.	
spring.mvc.contentnegotiation.parameter-name	Query parameter name to use when "favor-parameter" is enabled.	
spring.mvc.converters.preferred-json-mapper	Preferred JSON mapper to use for HTTP message conversion. By default, auto-detected according to the environment.	
spring.mvc.dispatch-options-request	Whether to dispatch OPTIONS requests to the FrameworkServlet doService method.	true
spring.mvc.dispatch-trace-request	Whether to dispatch TRACE requests to the FrameworkServlet doService method.	false

Name	Description	Default Value
spring.mvc.format.date	Date format to use, for example 'dd/MM/yyyy'.	
spring.mvc.format.date-time	Date-time format to use, for example 'yyyy-MM-dd HH:mm:ss'.	
spring.mvc.format.time	Time format to use, for example 'HH:mm:ss'.	
spring.mvc.formcontent.filter.enabled	Whether to enable Spring's FormContentFilter.	true
spring.mvc.hiddenmethod.filter.enabled	Whether to enable Spring's HiddenHttpMethodFilter.	false
spring.mvc.log-request-details	Whether logging of (potentially sensitive) request details at DEBUG and TRACE level is allowed.	false
spring.mvc.log-resolved-exception	Whether to enable warn logging of exceptions resolved by a "HandlerExceptionResolver", except for "DefaultHandlerExceptionResolver".	false
spring.mvc.message-codes-resolver-format	Formatting strategy for message codes. For instance, 'PREFIX_ERROR_CODE'.	
spring.mvc.pathmatch.matching-strategy	Choice of strategy for matching request paths against registered mappings.	path-pattern-parser
spring.mvc.problemdetails.enabled	Whether RFC 7807 Problem Details support should be enabled.	false
spring.mvc.publish-request-handled-events	Whether to publish a ServletRequestHandledEvent at the end of each request.	true
spring.mvc.servlet.load-on-startup	Load on startup priority of the dispatcher servlet.	-1

Name	Description	Default Value
spring.mvc.servlet.path	Path of the dispatcher servlet. Setting a custom value for this property is not compatible with the PathPatternParser matching strategy.	/
spring.mvc.static-path-pattern	Path pattern used for static resources.	/**
spring.mvc.throw-exception-if-no-handler-found	Whether a "NoHandlerFoundException " should be thrown if no Handler was found to process a request.	false
spring.mvc.view.prefix	Spring MVC view prefix.	
spring.mvc.view.suffix	Spring MVC view suffix.	
spring.mvc.webjars-path-pattern	Path pattern used for WebJar assets.	/webjars/**
spring.netty.leak-detection	Level of leak detection for reference-counted buffers. If not configured via 'ResourceLeakDetector.setLevel' or the 'io.netty.leakDetection.level' system property, default to 'simple'.	
spring.servlet.multipart.enabled	Whether to enable support of multipart uploads.	true
spring.servlet.multipart.file-size-threshold	Threshold after which files are written to disk.	0B
spring.servlet.multipart.location	Intermediate location of uploaded files.	
spring.servlet.multipart.max-file-size	Max file size.	1MB
spring.servlet.multipart.max-request-size	Max request size.	10MB
spring.servlet.multipart.resolve-lazily	Whether to resolve the multipart request lazily at the time of file or parameter access.	false

Name	Description	Default Value
spring.session.hazelcast.flush-mode	Sessions flush mode. Determines when session changes are written to the session store.	on-save
spring.session.hazelcast.map-name	Name of the map used to store sessions.	spring:session:sessions
spring.session.hazelcast.save-mode	Sessions save mode. Determines how session changes are tracked and saved to the session store.	on-set-attribute
spring.session.jdbc.cleanup-cron	Cron expression for expired session cleanup job.	0 * * * * *
spring.session.jdbc.flush-mode	Sessions flush mode. Determines when session changes are written to the session store.	on-save
spring.session.jdbc.initialize-schema	Database schema initialization mode.	embedded
spring.session.jdbc.platform	Platform to use in initialization scripts if the @@platform@@ placeholder is used. Auto-detected by default.	
spring.session.jdbc.save-mode	Sessions save mode. Determines how session changes are tracked and saved to the session store.	on-set-attribute
spring.session.jdbc.schema	Path to the SQL file to use to initialize the database schema.	classpath:org/springframework/session/jdbc/schema-@@platform@@.sql
spring.session.jdbc.table-name	Name of the database table used to store sessions.	SPRING_SESSION
spring.session.mongodb.collection-name	Collection name used to store sessions.	sessions
spring.session.redis.cleanup-cron	Cron expression for expired session cleanup job. Only supported when repository-type is set to indexed.	0 * * * * *

Name	Description	Default Value
spring.session.redis.configure-action	The configure action to apply when no user defined ConfigureRedisAction bean is present.	notify-keyspace-events
spring.session.redis.flush-mode	Sessions flush mode. Determines when session changes are written to the session store.	on-save
spring.session.redis.namespace	Namespace for keys used to store sessions.	spring:session
spring.session.redis.repository-type	Type of Redis session repository to configure.	default
spring.session.redis.save-mode	Sessions save mode. Determines how session changes are tracked and saved to the session store.	on-set-attribute
spring.session.servlet.filter-dispatcher-types	Session repository filter dispatcher types.	[async, error, request]
spring.session.servlet.filter-order	Session repository filter order.	
spring.session.timeout	Session timeout. If a duration suffix is not specified, seconds will be used.	
spring.web.locale	Locale to use. By default, this locale is overridden by the "Accept-Language" header.	
spring.web.locale-resolver	Define how the locale should be resolved.	accept-header
spring.web.resources.add-mappings	Whether to enable default resource handling.	true
spring.web.resources.cache.cachecontrol.cache-private	Indicate that the response message is intended for a single user and must not be stored by a shared cache.	
spring.web.resources.cache.cachecontrol.cache-public	Indicate that any cache may store the response.	
spring.web.resources.cache.cachecontrol.max-age	Maximum time the response should be cached, in seconds if no duration suffix is not specified.	

Name	Description	Default Value
spring.web.resources.cache.cachecontrol.must-revalidate	Indicate that once it has become stale, a cache must not use the response without re-validating it with the server.	
spring.web.resources.cache.cachecontrol.no-cache	Indicate that the cached response can be reused only if re-validated with the server.	
spring.web.resources.cache.cachecontrol.no-store	Indicate to not cache the response in any case.	
spring.web.resources.cache.cachecontrol.no-transform	Indicate intermediaries (caches and others) that they should not transform the response content.	
spring.web.resources.cache.cachecontrol.proxy-revalidate	Same meaning as the "must-revalidate" directive, except that it does not apply to private caches.	
spring.web.resources.cache.cachecontrol.s-max-age	Maximum time the response should be cached by shared caches, in seconds if no duration suffix is not specified.	
spring.web.resources.cache.cachecontrol.stale-if-error	Maximum time the response may be used when errors are encountered, in seconds if no duration suffix is not specified.	
spring.web.resources.cache.cachecontrol.stale-while-revalidate	Maximum time the response can be served after it becomes stale, in seconds if no duration suffix is not specified.	
spring.web.resources.cache.period	Cache period for the resources served by the resource handler. If a duration suffix is not specified, seconds will be used. Can be overridden by the 'spring.web.resources.cache.cachecontrol' properties.	

Name	Description	Default Value
spring.web.resources.cache.use-last-modified	Whether we should use the "lastModified" metadata of the files in HTTP caching headers.	true
spring.web.resources.chain.cache	Whether to enable caching in the Resource chain.	true
spring.web.resources.chain.compressed	Whether to enable resolution of already compressed resources (gzip, brotli). Checks for a resource name with the '.gz' or '.br' file extensions.	false
spring.web.resources.chain.enabled	Whether to enable the Spring Resource Handling chain. By default, disabled unless at least one strategy has been enabled.	
spring.web.resources.chain.strategy.content.enabled	Whether to enable the content Version Strategy.	false
spring.web.resources.chain.strategy.content.paths	Comma-separated list of patterns to apply to the content Version Strategy.	[/**]
spring.web.resources.chain.strategy.fixed.enabled	Whether to enable the fixed Version Strategy.	false
spring.web.resources.chain.strategy.fixed.paths	Comma-separated list of patterns to apply to the fixed Version Strategy.	[/**]
spring.web.resources.chain.strategy.fixed.version	Version string to use for the fixed Version Strategy.	
spring.web.resources.static-locations	Locations of static resources. Defaults to classpath:[/META-INF/resources/, /resources/, /static/, /public/].	[classpath:/META-INF/resources/, classpath:/resources/, classpath:/static/, classpath:/public/]
spring.webflux.base-path	Base path for all web handlers.	
spring.webflux.format.date	Date format to use, for example 'dd/MM/yyyy'.	
spring.webflux.format.date-time	Date-time format to use, for example 'yyyy-MM-dd HH:mm:ss'.	

Name	Description	Default Value
spring.webflux.format.time	Time format to use, for example 'HH:mm:ss'.	
spring.webflux.hiddenmethod.filter.enabled	Whether to enable Spring's HiddenHttpMethodFilter.	false
spring.webflux.multipart.file-storage-directory	Directory used to store file parts larger than 'maxInMemorySize'. Default is a directory named 'spring-multipart' created under the system temporary directory. Ignored when streaming is enabled.	
spring.webflux.multipart.headers-charset	Character set used to decode headers.	UTF-8
spring.webflux.multipart.max-disk-usage-per-part	Maximum amount of disk space allowed per part. Default is -1 which enforces no limits. Ignored when streaming is enabled.	-1B
spring.webflux.multipart.max-headers-size	Maximum amount of memory allowed per headers section of each part. Set to -1 to enforce no limits.	10KB
spring.webflux.multipart.max-in-memory-size	Maximum amount of memory allowed per part before it's written to disk. Set to -1 to store all contents in memory. Ignored when streaming is enabled.	256KB
spring.webflux.multipart.max-parts	Maximum number of parts allowed in a given multipart request. Default is -1 which enforces no limits.	-1
spring.webflux.problemdetails.enabled	Whether RFC 7807 Problem Details support should be enabled.	false
spring.webflux.static-path-pattern	Path pattern used for static resources.	/**
spring.webflux.webjars-path-pattern	Path pattern used for WebJar assets.	/webjars/**

.A.10. Templating Properties

Name	Description	Default Value
spring.freemarker.allow-request-override	Whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.	false
spring.freemarker.allow-session-override	Whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.	false
spring.freemarker.cache	Whether to enable template caching.	false
spring.freemarker.charset	Template encoding.	UTF-8
spring.freemarker.check-template-location	Whether to check that the templates location exists.	true
spring.freemarker.content-type	Content-Type value.	text/html
spring.freemarker.enabled	Whether to enable MVC view resolution for this technology.	true
spring.freemarker.expose-request-attributes	Whether all request attributes should be added to the model prior to merging with the template.	false
spring.freemarker.expose-session-attributes	Whether all HttpSession attributes should be added to the model prior to merging with the template.	false
spring.freemarker.expose-spring-macro-helpers	Whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".	true

Name	Description	Default Value
spring.freemarker.prefer-file-system-access	Whether to prefer file system access for template loading to enable hot detection of template changes. When a template path is detected as a directory, templates are loaded from the directory only and other matching classpath locations will not be considered.	false
spring.freemarker.prefix	Prefix that gets prepended to view names when building a URL.	
spring.freemarker.request-context-attribute	Name of the RequestContext attribute for all views.	
spring.freemarker.settings.*	Well-known FreeMarker keys which are passed to FreeMarker's Configuration.	
spring.freemarker.suffix	Suffix that gets appended to view names when building a URL.	.ftlh
spring.freemarker.template-loader-path	Comma-separated list of template paths.	[classpath:/templates/]
spring.freemarker.view-names	View names that can be resolved.	
spring.groovy.template.allow-request-override	Whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.	false
spring.groovy.template.allow-session-override	Whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.	false
spring.groovy.template.cache	Whether to enable template caching.	false
spring.groovy.template.charset	Template encoding.	UTF-8
spring.groovy.template.check-template-location	Whether to check that the templates location exists.	true

Name	Description	Default Value
spring.groovy.template.configuration.auto-escape spring.groovy.template.configuration.auto-indent spring.groovy.template.configuration.auto-indent-string spring.groovy.template.configuration.auto-new-line spring.groovy.template.configuration.base-template-class spring.groovy.template.configuration.cache-templates spring.groovy.template.configuration.declaration-encoding spring.groovy.template.configuration.expand-empty-elements spring.groovy.template.configuration.locale spring.groovy.template.configuration.new-line-string spring.groovy.template.configuration.resource-loader-path spring.groovy.template.configuration.use-double-quotes	See GroovyMarkupConfigurer	
spring.groovy.template.content-type	Content-Type value.	text/html
spring.groovy.template.enabled	Whether to enable MVC view resolution for this technology.	true
spring.groovy.template.expose-request-attributes	Whether all request attributes should be added to the model prior to merging with the template.	false
spring.groovy.template.expose-session-attributes	Whether all HttpSession attributes should be added to the model prior to merging with the template.	false
spring.groovy.template.expose-spring-macro-helpers	Whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".	true

Name	Description	Default Value
spring.groovy.template.prefix	Prefix that gets prepended to view names when building a URL.	
spring.groovy.template.request-context-attribute	Name of the RequestContext attribute for all views.	
spring.groovy.template.resource-loader-path	Template path.	classpath:/templates/
spring.groovy.template.suffix	Suffix that gets appended to view names when building a URL.	.tpl
spring.groovy.template.view-names	View names that can be resolved.	
spring.mustache.charset	Template encoding.	UTF-8
spring.mustache.check-template-location	Whether to check that the templates location exists.	true
spring.mustache.enabled	Whether to enable MVC view resolution for Mustache.	true
spring.mustache.prefix	Prefix to apply to template names.	classpath:/templates/
spring.mustache.reactive.media-types	Media types supported by Mustache views.	text/html;charset=UTF-8
spring.mustache.request-context-attribute	Name of the RequestContext attribute for all views.	
spring.mustache.servlet.allow-request-override	Whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.	false
spring.mustache.servlet.allow-session-override	Whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.	false
spring.mustache.servlet.cache	Whether to enable template caching.	false
spring.mustache.servlet.content-type	Content-Type value.	

Name	Description	Default Value
spring.mustache.servlet.expose-request-attributes	Whether all request attributes should be added to the model prior to merging with the template.	false
spring.mustache.servlet.expose-session-attributes	Whether all HttpSession attributes should be added to the model prior to merging with the template.	false
spring.mustache.servlet.expose-spring-macro-helpers	Whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".	true
spring.mustache.suffix	Suffix to apply to template names.	.mustache
spring.mustache.view-names	View names that can be resolved.	
spring.thymeleaf.cache	Whether to enable template caching.	true
spring.thymeleaf.check-template	Whether to check that the template exists before rendering it.	true
spring.thymeleaf.check-template-location	Whether to check that the templates location exists.	true
spring.thymeleaf.enable-spring-el-compiler	Enable the SpringEL compiler in SpringEL expressions.	false
spring.thymeleaf.enabled	Whether to enable Thymeleaf view resolution for Web frameworks.	true
spring.thymeleaf.encoding	Template files encoding.	UTF-8
spring.thymeleaf.excluded-view-names	Comma-separated list of view names (patterns allowed) that should be excluded from resolution.	
spring.thymeleaf.mode	Template mode to be applied to templates. See also Thymeleaf's TemplateMode enum.	HTML

Name	Description	Default Value
spring.thymeleaf.prefix	Prefix that gets prepended to view names when building a URL.	classpath:/templates/
spring.thymeleaf.reactive.chunked-mode-view-names	Comma-separated list of view names (patterns allowed) that should be the only ones executed in CHUNKED mode when a max chunk size is set.	
spring.thymeleaf.reactive.full-mode-view-names	Comma-separated list of view names (patterns allowed) that should be executed in FULL mode even if a max chunk size is set.	
spring.thymeleaf.reactive.max-chunk-size	Maximum size of data buffers used for writing to the response. Templates will execute in CHUNKED mode by default if this is set.	0B
spring.thymeleaf.reactive.media-types	Media types supported by the view technology.	[text/html, application/xhtml+xml, application/xml, text/xml, application/rss+xml, application/atom+xml, application/javascript, application/ecmascript, text/javascript, text/ecmascript, application/json, text/css, text/plain, text/event-stream]
spring.thymeleaf.render-hidden-markers-before-checkboxes	Whether hidden form inputs acting as markers for checkboxes should be rendered before the checkbox element itself.	false
spring.thymeleaf.servlet.content-type	Content-Type value written to HTTP responses.	text/html
spring.thymeleaf.servlet.produce-partial-output-while-processing	Whether Thymeleaf should start writing partial output as soon as possible or buffer until template processing is finished.	true

Name	Description	Default Value
spring.thymeleaf.suffix	Suffix that gets appended to view names when building a URL.	.html
spring.thymeleaf.template-resolver-order	Order of the template resolver in the chain. By default, the template resolver is first in the chain. Order start at 1 and should only be set if you have defined additional "TemplateResolver" beans.	
spring.thymeleaf.view-names	Comma-separated list of view names (patterns allowed) that can be resolved.	

.A.11. Server Properties

Name	Description	Default Value
server.address	Network address to which the server should bind.	
server.compression.enabled	Whether response compression is enabled.	false
server.compression.excluded-user-agents	Comma-separated list of user agents for which responses should not be compressed.	
server.compression.mime-types	Comma-separated list of MIME types that should be compressed.	[text/html, text/xml, text/plain, text/css, text/javascript, application/javascript, application/json, application/xml]
server.compression.min-response-size	Minimum "Content-Length" value that is required for compression to be performed.	2KB
server.error.include-binding-errors	When to include "errors" attribute.	never
server.error.include-exception	Include the "exception" attribute.	false
server.error.include-message	When to include "message" attribute.	never

Name	Description	Default Value
server.error.include-stacktrace	When to include the "trace" attribute.	never
server.error.path	Path of the error controller.	/error
server.error.whitelabel.enabled	Whether to enable the default error page displayed in browsers in case of a server error.	true
server.forward-headers-strategy	Strategy for handling X-Forwarded-* headers.	
server.http2.enabled	Whether to enable HTTP/2 support, if the current environment supports it.	false
server.jetty.accesslog.append	Append to log.	false
server.jetty.accesslog.custom-format	Custom log format, see org.eclipse.jetty.server.CustomRequestLog. If defined, overrides the "format" configuration key.	
server.jetty.accesslog.enabled	Enable access log.	false
server.jetty.accesslog.file-date-format	Date format to place in log file name.	
server.jetty.accesslog.filename	Log filename. If not specified, logs redirect to "System.err".	
server.jetty.accesslog.format	Log format.	ncsa
server.jetty.accesslog.ignore-paths	Request paths that should not be logged.	
server.jetty.accesslog.retention-period	Number of days before rotated log files are deleted.	31
server.jetty.connection-idle-timeout	Time that the connection can be idle before it is closed.	
server.jetty.max-http-form-post-size	Maximum size of the form content in any HTTP post request.	200000B
server.jetty.max-http-response-header-size	Maximum size of the HTTP response header.	8KB

Name	Description	Default Value
server.jetty.threads.acceptors	Number of acceptor threads to use. When the value is -1, the default, the number of acceptors is derived from the operating environment.	-1
server.jetty.threads.idle-timeout	Maximum thread idle time.	60000ms
server.jetty.threads.max	Maximum number of threads.	200
server.jetty.threads.max-queue-capacity	Maximum capacity of the thread pool's backing queue. A default is computed based on the threading configuration.	
server.jetty.threads.min	Minimum number of threads.	8
server.jetty.threads.selectors	Number of selector threads to use. When the value is -1, the default, the number of selectors is derived from the operating environment.	-1
server.max-http-request-header-size	Maximum size of the HTTP request header.	8KB
server.netty.connection-timeout	Connection timeout of the Netty channel.	
server.netty.h2c-max-content-length	Maximum content length of an H2C upgrade request.	0B
server.netty.idle-timeout	Idle timeout of the Netty channel. When not specified, an infinite timeout is used.	
server.netty.initial-buffer-size	Initial buffer size for HTTP request decoding.	128B
server.netty.max-initial-line-length	Maximum length that can be decoded for an HTTP request's initial line.	4KB
server.netty.max-keep-alive-requests	Maximum number of requests that can be made per connection. By default, a connection serves unlimited number of requests.	

Name	Description	Default Value
server.netty.validate-headers	Whether to validate headers when decoding requests.	true
server.port	Server HTTP port.	8080
server.reactive.session.cookie.domain	Domain for the cookie.	
server.reactive.session.cookie.http-only	Whether to use "HttpOnly" cookies for the cookie.	
server.reactive.session.cookie.max-age	Maximum age of the cookie. If a duration suffix is not specified, seconds will be used. A positive value indicates when the cookie expires relative to the current time. A value of 0 means the cookie should expire immediately. A negative value means no "Max-Age".	
server.reactive.session.cookie.name	Name for the cookie.	
server.reactive.session.cookie.path	Path of the cookie.	
server.reactive.session.cookie.same-site	SameSite setting for the cookie.	
server.reactive.session.cookie.secure	Whether to always mark the cookie as secure.	
server.reactive.session.timeout	Session timeout. If a duration suffix is not specified, seconds will be used.	30m
server.server-header	Value to use for the Server response header (if empty, no header is sent).	
server.servlet.application-display-name	Display name of the application.	application
server.servlet.context-parameters.*	Servlet context init parameters.	
server.servlet.context-path	Context path of the application.	
server.servlet.encoding.charset	Charset of HTTP requests and responses. Added to the "Content-Type" header if not set explicitly.	UTF-8

Name	Description	Default Value
server.servlet.encoding.enabled	Whether to enable http encoding support.	true
server.servlet.encoding.force	Whether to force the encoding to the configured charset on HTTP requests and responses.	
server.servlet.encoding.force-request	Whether to force the encoding to the configured charset on HTTP requests. Defaults to true when "force" has not been specified.	
server.servlet.encoding.force-response	Whether to force the encoding to the configured charset on HTTP responses.	
server.servlet.encoding.mapping.*	Mapping of locale to charset for response encoding.	
server.servlet.jsp.class-name	Class name of the servlet to use for JSPs. If registered is true and this class * is on the classpath then it will be registered.	org.apache.jasper.servlet.JspServlet
server.servlet.jsp.init-parameters.*	Init parameters used to configure the JSP servlet.	
server.servlet.jsp.registered	Whether the JSP servlet is registered.	true
server.servlet.register-default-servlet	Whether to register the default Servlet with the container.	false
server.servlet.session.cookie.domain	Domain for the cookie.	
server.servlet.session.cookie.http-only	Whether to use "HttpOnly" cookies for the cookie.	

Name	Description	Default Value
server.servlet.session.cookie.max-age	Maximum age of the cookie. If a duration suffix is not specified, seconds will be used. A positive value indicates when the cookie expires relative to the current time. A value of 0 means the cookie should expire immediately. A negative value means no "Max-Age".	
server.servlet.session.cookie.name	Name of the cookie.	
server.servlet.session.cookie.path	Path of the cookie.	
server.servlet.session.cookie.same-site	SameSite setting for the cookie.	
server.servlet.session.cookie.secure	Whether to always mark the cookie as secure.	
server.servlet.session.persistent	Whether to persist session data between restarts.	false
server.servlet.session.store-dir	Directory used to store session data.	
server.servlet.session.timeout	Session timeout. If a duration suffix is not specified, seconds will be used.	30m
server.servlet.session.tracking-modes	Session tracking modes.	
server.shutdown	Type of shutdown that the server will support.	immediate
server.ssl.bundle	The name of a configured SSL bundle.	
server.ssl.certificate	Path to a PEM-encoded SSL certificate file.	
server.ssl.certificate-private-key	Path to a PEM-encoded private key file for the SSL certificate.	
server.ssl.ciphers	Supported SSL ciphers.	
server.ssl.client-auth	Client authentication mode. Requires a trust store.	
server.ssl.enabled	Whether to enable SSL support.	true

Name	Description	Default Value
server.ssl.enabled-protocols	Enabled SSL protocols.	
server.ssl.key-alias	Alias that identifies the key in the key store.	
server.ssl.key-password	Password used to access the key in the key store.	
server.ssl.key-store	Path to the key store that holds the SSL certificate (typically a jks file).	
server.ssl.key-store-password	Password used to access the key store.	
server.ssl.key-store-provider	Provider for the key store.	
server.ssl.key-store-type	Type of the key store.	
server.ssl.protocol	SSL protocol to use.	TLS
server.ssl.trust-certificate	Path to a PEM-encoded SSL certificate authority file.	
server.ssl.trust-certificate-private-key	Path to a PEM-encoded private key file for the SSL certificate authority.	
server.ssl.trust-store	Trust store that holds SSL certificates.	
server.ssl.trust-store-password	Password used to access the trust store.	
server.ssl.trust-store-provider	Provider for the trust store.	
server.ssl.trust-store-type	Type of the trust store.	
server.tomcat.accept-count	Maximum queue length for incoming connection requests when all possible request processing threads are in use.	100
server.tomcat.accesslog.buffered	Whether to buffer output such that it is flushed only periodically.	true
server.tomcat.accesslog.check-exists	Whether to check for log file existence so it can be recreated if an external process has renamed it.	false

Name	Description	Default Value
server.tomcat.accesslog.condition-if	Whether logging of the request will only be enabled if "ServletRequest.getAttribute (conditionIf)" does not yield null.	
server.tomcat.accesslog.condition-unless	Whether logging of the request will only be enabled if "ServletRequest.getAttribute (conditionUnless)" yield null.	
server.tomcat.accesslog.directory	Directory in which log files are created. Can be absolute or relative to the Tomcat base dir.	logs
server.tomcat.accesslog.enabled	Enable access log.	false
server.tomcat.accesslog.encoding	Character set used by the log file. Default to the system default character set.	
server.tomcat.accesslog.file-date-format	Date format to place in the log file name.	.yyyy-MM-dd
server.tomcat.accesslog.ipv6-canonical	Whether to use IPv6 canonical representation format as defined by RFC 5952.	false
server.tomcat.accesslog.locale	Locale used to format timestamps in log entries and in log file name suffix. Default to the default locale of the Java process.	
server.tomcat.accesslog.max-days	Number of days to retain the access log files before they are removed.	-1
server.tomcat.accesslog.pattern	Format pattern for access logs.	common
server.tomcat.accesslog.prefix	Log file name prefix.	access_log
server.tomcat.accesslog.rename-on-rotate	Whether to defer inclusion of the date stamp in the file name until rotate time.	false

Name	Description	Default Value
server.tomcat.accesslog.request-attributes-enabled	Set request attributes for the IP address, Hostname, protocol, and port used for the request.	false
server.tomcat.accesslog.rotate	Whether to enable access log rotation.	true
server.tomcat.accesslog.suffix	Log file name suffix.	.log
server.tomcat.additional-tld-skip-patterns	Comma-separated list of additional patterns that match jars to ignore for TLD scanning. The special '?' and '*' characters can be used in the pattern to match one and only one character and zero or more characters respectively.	
server.tomcat.background-processor-delay	Delay between the invocation of backgroundProcess methods. If a duration suffix is not specified, seconds will be used.	10s
server.tomcat.basedir	Tomcat base directory. If not specified, a temporary directory is used.	
server.tomcat.connection-timeout	Amount of time the connector will wait, after accepting a connection, for the request URI line to be presented.	
server.tomcat.keep-alive-timeout	Time to wait for another HTTP request before the connection is closed. When not set the connectionTimeout is used. When set to -1 there will be no timeout.	

Name	Description	Default Value
server.tomcat.max-connections	Maximum number of connections that the server accepts and processes at any given time. Once the limit has been reached, the operating system may still accept connections based on the "acceptCount" property.	8192
server.tomcat.max-http-form-post-size	Maximum size of the form content in any HTTP post request.	2MB
server.tomcat.max-http-response-header-size	Maximum size of the HTTP response header.	8KB
server.tomcat.max-keep-alive-requests	Maximum number of HTTP requests that can be pipelined before the connection is closed. When set to 0 or 1, keep-alive and pipelining are disabled. When set to -1, an unlimited number of pipelined or keep-alive requests are allowed.	100
server.tomcat.max-swallow-size	Maximum amount of request body to swallow.	2MB
server.tomcat.mbeanregistry.enabled	Whether Tomcat's MBean Registry should be enabled.	false
server.tomcat.processor-cache	Maximum number of idle processors that will be retained in the cache and reused with a subsequent request. When set to -1 the cache will be unlimited with a theoretical maximum size equal to the maximum number of connections.	200
server.tomcat.redirect-context-root	Whether requests to the context root should be redirected by appending a / to the path. When using SSL terminated at a proxy, this property should be set to false.	true

Name	Description	Default Value
server.tomcat.relaxed-path-chars	Comma-separated list of additional unencoded characters that should be allowed in URI paths. Only "< > [\ ] ^ ` { | }" are allowed.	
server.tomcat.relaxed-query-chars	Comma-separated list of additional unencoded characters that should be allowed in URI query strings. Only "< > [\ ] ^ ` { | }" are allowed.	
server.tomcat.remoteip.host-header	Name of the HTTP header from which the remote host is extracted.	X-Forwarded-Host
server.tomcat.remoteip.internal-proxies	Regular expression that matches proxies that are to be trusted.	10\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|192\\.168\\.\\d{1,3}\\.\\d{1,3}|169\\.254\\.\\d{1,3}\\.\\d{1,3}|127\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|100\\.6[4-9]{1}\\.\\d{1,3}\\.\\d{1,3}|100\\.[7-9]{1}\\d{1}\\.\\d{1,3}\\.\\d{1,3}|100\\.1[0-1]{1}\\d{1}\\.\\d{1,3}\\.\\d{1,3}|100\\.12[0-7]{1}\\.\\d{1,3}\\.\\d{1,3}|172\\.1[6-9]{1}\\.\\d{1,3}\\.\\d{1,3}|172\\.2[0-9]{1}\\.\\d{1,3}\\.\\d{1,3}|172\\.3[0-1]{1}\\.\\d{1,3}\\.\\d{1,3}|0:0:0:0:0:0:0:1|::1
server.tomcat.remoteip.port-header	Name of the HTTP header used to override the original port value.	X-Forwarded-Port
server.tomcat.remoteip.protocol-header	Header that holds the incoming protocol, usually named "X-Forwarded-Proto".	
server.tomcat.remoteip.protocol-header-https-value	Value of the protocol header indicating whether the incoming request uses SSL.	https

Name	Description	Default Value
server.tomcat.remoteip.remote-ip-header	Name of the HTTP header from which the remote IP is extracted. For instance, 'X-FORWARDED-FOR'.	
server.tomcat.remoteip.trusted-proxies	Regular expression defining proxies that are trusted when they appear in the "remote-ip-header" header.	
server.tomcat.resource.allow-caching	Whether static resource caching is permitted for this web application.	true
server.tomcat.resource.cache-ttl	Time-to-live of the static resource cache.	
server.tomcat.threads.max	Maximum amount of worker threads.	200
server.tomcat.threads.min-spare	Minimum amount of worker threads.	10
server.tomcat.uri-encoding	Character encoding to use to decode the URI.	UTF-8
server.tomcat.use-relative-redirects	Whether HTTP 1.1 and later location headers generated by a call to sendRedirect will use relative or absolute redirects.	false
server.undertow.accesslog.dir	Undertow access log directory.	
server.undertow.accesslog.enabled	Whether to enable the access log.	false
server.undertow.accesslog.pattern	Format pattern for access logs.	common
server.undertow.accesslog.prefix	Log file name prefix.	access_log.
server.undertow.accesslog.rotate	Whether to enable access log rotation.	true
server.undertow.accesslog.suffix	Log file name suffix.	log
server.undertow.always-set-keep-alive	Whether the 'Connection: keep-alive' header should be added to all responses, even if not required by the HTTP specification.	true

Name	Description	Default Value
server.undertow.buffer-size	Size of each buffer. The default is derived from the maximum amount of memory that is available to the JVM.	
server.undertow.decode-slash	Whether encoded slash characters (%2F) should be decoded. Decoding can cause security problems if a front-end proxy does not perform the same decoding. Only enable this if you have a legacy application that requires it. When set, server.undertow.allow-encoded-slash has no effect.	
server.undertow.decode-url	Whether the URL should be decoded. When disabled, percent-encoded characters in the URL will be left as-is.	true
server.undertow.direct-buffers	Whether to allocate buffers outside the Java heap. The default is derived from the maximum amount of memory that is available to the JVM.	
server.undertow.eager-filter-init	Whether servlet filters should be initialized on startup.	true
server.undertow.max-cookies	Maximum number of cookies that are allowed. This limit exists to prevent hash collision based DOS attacks.	200
server.undertow.max-headers	Maximum number of headers that are allowed. This limit exists to prevent hash collision based DOS attacks.	
server.undertow.max-http-post-size	Maximum size of the HTTP post content. When the value is -1, the default, the size is unlimited.	-1B

Name	Description	Default Value
server.undertow.max-parameters	Maximum number of query or path parameters that are allowed. This limit exists to prevent hash collision based DOS attacks.	
server.undertow.no-request-timeout	Amount of time a connection can sit idle without processing a request, before it is closed by the server.	
server.undertow.options.server.*	Server options as defined in io.undertow.UndertowOptions.	
server.undertow.options.socket.*	Socket options as defined in org.xnio.Options.	
server.undertow.preserve-path-on-forward	Whether to preserve the path of a request when it is forwarded.	false
server.undertow.threads.io	Number of I/O threads to create for the worker. The default is derived from the number of available processors.	
server.undertow.threads.worker	Number of worker threads. The default is 8 times the number of I/O threads.	
server.undertow.url-charset	Charset used to decode URLs.	UTF-8

.A.12. Security Properties

Name	Description	Default Value
spring.security.filter.dispatcher-types	Security filter chain dispatcher types for Servlet-based web applications.	[async, error, forward, include, request]
spring.security.filter.order	Security filter chain order for Servlet-based web applications.	-100
spring.security.oauth2.authorizationserver.client.*	Registered clients of the Authorization Server.	

Name	Description	Default Value
spring.security.oauth2.authorizationserver.endpoint.authorization-uri	Authorization Server's OAuth 2.0 Authorization Endpoint.	/oauth2/authorize
spring.security.oauth2.authorizationserver.endpoint.device-authorization-uri	Authorization Server's OAuth 2.0 Device Authorization Endpoint.	/oauth2/device_authorization
spring.security.oauth2.authorizationserver.endpoint.device-verification-uri	Authorization Server's OAuth 2.0 Device Verification Endpoint.	/oauth2/device_verification
spring.security.oauth2.authorizationserver.endpoint.jwk-set-uri	Authorization Server's JWK Set Endpoint.	/oauth2/jwks
spring.security.oauth2.authorizationserver.endpoint.oidc.client-registration-uri	Authorization Server's OpenID Connect 1.0 Client Registration Endpoint.	/connect/register
spring.security.oauth2.authorizationserver.endpoint.oidc.logout-uri	Authorization Server's OpenID Connect 1.0 Logout Endpoint.	/connect/logout
spring.security.oauth2.authorizationserver.endpoint.oidc.user-info-uri	Authorization Server's OpenID Connect 1.0 UserInfo Endpoint.	/userinfo
spring.security.oauth2.authorizationserver.endpoint.token-introspection-uri	Authorization Server's OAuth 2.0 Token Introspection Endpoint.	/oauth2/introspect
spring.security.oauth2.authorizationserver.endpoint.token-revocation-uri	Authorization Server's OAuth 2.0 Token Revocation Endpoint.	/oauth2/revoke
spring.security.oauth2.authorizationserver.endpoint.token-uri	Authorization Server's OAuth 2.0 Token Endpoint.	/oauth2/token
spring.security.oauth2.authorizationserver.issuer	URL of the Authorization Server's Issuer Identifier.	
spring.security.oauth2.client.provider.*	OAuth provider details.	
spring.security.oauth2.client.registration.*	OAuth client registrations.	
spring.security.oauth2.resourceserver.jwt.audiences	Identifies the recipients that the JWT is intended for.	

Name	Description	Default Value
spring.security.oauth2.resourceserver.jwt.issuer-uri	URI that can either be an OpenID Connect discovery endpoint or an OAuth 2.0 Authorization Server Metadata endpoint defined by RFC 8414.	
spring.security.oauth2.resourceserver.jwt.jwk-set-uri	JSON Web Key URI to use to verify the JWT token.	
spring.security.oauth2.resourceserver.jwt.jws-algorithms	JSON Web Algorithms used for verifying the digital signatures.	RS256
spring.security.oauth2.resourceserver.jwt.public-key-location	Location of the file containing the public key used to verify a JWT.	
spring.security.oauth2.resourceserver.opaquetoken.client-id	Client id used to authenticate with the token introspection endpoint.	
spring.security.oauth2.resourceserver.opaquetoken.client-secret	Client secret used to authenticate with the token introspection endpoint.	
spring.security.oauth2.resourceserver.opaquetoken.introspection-uri	OAuth 2.0 endpoint through which token introspection is accomplished.	
spring.security.saml2.relyingparty.registration.*	SAML2 relying party registrations.	
spring.security.user.name	Default user name.	user
spring.security.user.password	Password for the default user name.	
spring.security.user.roles	Granted roles for the default user name.	

.A.13. RSocket Properties

Name	Description	Default Value
spring.rsocket.server.address	Network address to which the server should bind.	
spring.rsocket.server.fragment-size	Maximum transmission unit. Frames larger than the specified value are fragmented.	

Name	Description	Default Value
spring.rsocket.server.mapping-path	Path under which RSocket handles requests (only works with websocket transport).	
spring.rsocket.server.port	Server port.	
spring.rsocket.server.ssl.bundle	The name of a configured SSL bundle.	
spring.rsocket.server.ssl.certificate	Path to a PEM-encoded SSL certificate file.	
spring.rsocket.server.ssl.certificate-private-key	Path to a PEM-encoded private key file for the SSL certificate.	
spring.rsocket.server.ssl.ciphers	Supported SSL ciphers.	
spring.rsocket.server.ssl.client-auth	Client authentication mode. Requires a trust store.	
spring.rsocket.server.ssl.enabled	Whether to enable SSL support.	true
spring.rsocket.server.ssl.enabled-protocols	Enabled SSL protocols.	
spring.rsocket.server.ssl.key-alias	Alias that identifies the key in the key store.	
spring.rsocket.server.ssl.key-password	Password used to access the key in the key store.	
spring.rsocket.server.ssl.key-store	Path to the key store that holds the SSL certificate (typically a jks file).	
spring.rsocket.server.ssl.key-store-password	Password used to access the key store.	
spring.rsocket.server.ssl.key-store-provider	Provider for the key store.	
spring.rsocket.server.ssl.key-store-type	Type of the key store.	
spring.rsocket.server.ssl.protocol	SSL protocol to use.	TLS
spring.rsocket.server.ssl.trust-certificate	Path to a PEM-encoded SSL certificate authority file.	
spring.rsocket.server.ssl.trust-certificate-private-key	Path to a PEM-encoded private key file for the SSL certificate authority.	
spring.rsocket.server.ssl.trust-store	Trust store that holds SSL certificates.	

Name	Description	Default Value
spring.rsocket.server.ssl.trust-store-password	Password used to access the trust store.	
spring.rsocket.server.ssl.trust-store-provider	Provider for the trust store.	
spring.rsocket.server.ssl.trust-store-type	Type of the trust store.	
spring.rsocket.server.transport	RSocket transport protocol.	tcp

.A.14. Actuator Properties

Name	Description	Default Value
management.appoptics.metrics.export.api-token	AppOptics API token.	
management.appoptics.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	500
management.appoptics.metrics.export.connect-timeout	Connection timeout for requests to this backend.	5s
management.appoptics.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.appoptics.metrics.export.floor-times	Whether to ship a floored time, useful when sending measurements from multiple hosts to align them on a given time boundary.	false
management.appoptics.metrics.export.host-tag	Tag that will be mapped to "@host" when shipping metrics to AppOptics.	instance
management.appoptics.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.appoptics.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.appoptics.metrics.export.uri	URI to ship metrics to.	https://api.appoptics.com/v1/measurements

Name	Description	Default Value
management.atlas.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.atlas.metrics.export.config-refresh-frequency	Frequency for refreshing config settings from the LWC service.	10s
management.atlas.metrics.export.config-time-to-live	Time to live for subscriptions from the LWC service.	150s
management.atlas.metrics.export.config-uri	URI for the Atlas LWC endpoint to retrieve current subscriptions.	http://localhost:7101/lwc/api/v1/expressions/local-dev
management.atlas.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.atlas.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.atlas.metrics.export.eval-uri	URI for the Atlas LWC endpoint to evaluate the data for a subscription.	http://localhost:7101/lwc/api/v1/evaluate
management.atlas.metrics.export.lwc-enabled	Whether to enable streaming to Atlas LWC.	false
management.atlas.metrics.export.lwc-ignore-publish-step	Whether expressions with the same step size as Atlas publishing should be ignored for streaming. Used for cases where data being published to Atlas is also sent into streaming from the backend.	true

Name	Description	Default Value
management.atlas.metrics.export.lwc-step	Step size (reporting frequency) to use for streaming to Atlas LWC. This is the highest supported resolution for getting an on-demand stream of the data. It must be less than or equal to management.metrics.export.atlas.step and management.metrics.export.atlas.step should be an even multiple of this value.	5s
management.atlas.metrics.export.meter-time-to-live	Time to live for meters that do not have any activity. After this period the meter will be considered expired and will not get reported.	15m
management.atlas.metrics.export.num-threads	Number of threads to use with the metrics publishing scheduler.	4
management.atlas.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.atlas.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.atlas.metrics.export.uri	URI of the Atlas server.	http://localhost:7101/api/v1/publish
management.auditevents.enabled	Whether to enable storage of audit events.	true
management.cloudfoundry.enabled	Whether to enable extended Cloud Foundry actuator endpoints.	true
management.cloudfoundry.skip-ssl-validation	Whether to skip SSL verification for Cloud Foundry actuator endpoint security calls.	false
management.datadog.metrics.export.api-key	Datadog API key.	

Name	Description	Default Value
management.datadog.metrics.export.application-key	Datadog application key. Not strictly required, but improves the Datadog experience by sending meter descriptions, types, and base units to Datadog.	
management.datadog.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.datadog.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.datadog.metrics.export.descriptions	Whether to publish descriptions metadata to Datadog. Turn this off to minimize the amount of metadata sent.	true
management.datadog.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.datadog.metrics.export.host-tag	Tag that will be mapped to "host" when shipping metrics to Datadog.	instance
management.datadog.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.datadog.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.datadog.metrics.export.uri	URI to ship metrics to. Set this if you need to publish metrics to a Datadog site other than US, or to an internal proxy en-route to Datadog.	https://api.datadoghq.com
management.defaults.metrics.export.enabled	Whether to enable default metrics exporters.	true
management.dynatrace.metrics.export.api-token	Dynatrace authentication token.	

Name	Description	Default Value
management.dynatrace.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.dynatrace.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.dynatrace.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.dynatrace.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.dynatrace.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.dynatrace.metrics.export.uri	URI to ship metrics to. Should be used for SaaS, self-managed instances or to en-route through an internal proxy.	
management.dynatrace.metrics.export.v1.device-id	ID of the custom device that is exporting metrics to Dynatrace.	
management.dynatrace.metrics.export.v1.group	Group for exported metrics. Used to specify custom device group name in the Dynatrace UI.	
management.dynatrace.metrics.export.v1.technology-type	Technology type for exported metrics. Used to group metrics under a logical technology name in the Dynatrace UI.	java
management.dynatrace.metrics.export.v2.default-dimensions.*	Default dimensions that are added to all metrics in the form of key-value pairs. These are overwritten by Micrometer tags if they use the same key.	
management.dynatrace.metrics.export.v2.enrich-with-dynatrace-metadata	Whether to enable Dynatrace metadata export.	true
management.dynatrace.metrics.export.v2.metric-key-prefix	Prefix string that is added to all exported metrics.	

Name	Description	Default Value
management.dynatrace.metrics.export.v2.use-dynatrace-summary-instruments	Whether to fall back to the built-in micrometer instruments for Timer and DistributionSummary.	true
management.elastic.metrics.export.api-key-credentials	Base64-encoded credentials string. Mutually exclusive with user-name and password.	
management.elastic.metrics.export.auto-create-index	Whether to create the index automatically if it does not exist.	true
management.elastic.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.elastic.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.elastic.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.elastic.metrics.export.host	Host to export metrics to.	http://localhost:9200
management.elastic.metrics.export.index	Index to export metrics to.	micrometer-metrics
management.elastic.metrics.export.index-date-format	Index date format used for rolling indices. Appended to the index name.	yyyy-MM
management.elastic.metrics.export.index-date-separator	Prefix to separate the index name from the date format used for rolling indices.	-
management.elastic.metrics.export.password	Login password of the Elastic server. Mutually exclusive with api-key-credentials.	
management.elastic.metrics.export.pipeline	Ingest pipeline name. By default, events are not pre-processed.	
management.elastic.metrics.export.read-timeout	Read timeout for requests to this backend.	10s

Name	Description	Default Value
management.elastic.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.elastic.metrics.export.timestamp-field-name	Name of the timestamp field.	@timestamp
management.elastic.metrics.export.user-name	Login user of the Elastic server. Mutually exclusive with api-key-credentials.	
management.endpoint.auditevents.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.auditevents.enabled	Whether to enable the auditevents endpoint.	true
management.endpoint.beans.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.beans.enabled	Whether to enable the beans endpoint.	true
management.endpoint.caches.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.caches.enabled	Whether to enable the caches endpoint.	true
management.endpoint.conditions.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.conditions.enabled	Whether to enable the conditions endpoint.	true
management.endpoint.configprops.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.configprops.enabled	Whether to enable the configprops endpoint.	true
management.endpoint.configprops.roles	Roles used to determine whether a user is authorized to be shown unsanitized values. When empty, all authenticated users are authorized.	
management.endpoint.configprops.show-values	When to show unsanitized values.	
management.endpoint.env.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.env.enabled	Whether to enable the env endpoint.	true

Name	Description	Default Value
management.endpoint.env.roles	Roles used to determine whether a user is authorized to be shown unsanitized values. When empty, all authenticated users are authorized.	
management.endpoint.env.show-values	When to show unsanitized values.	
management.endpoint.flyway.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.flyway.enabled	Whether to enable the flyway endpoint.	true
management.endpoint.health.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.health.enabled	Whether to enable the health endpoint.	true
management.endpoint.health.group.*	Health endpoint groups.	
management.endpoint.health.logging.slow-indicator-threshold	Threshold after which a warning will be logged for slow health indicators.	10s
management.endpoint.health.probes.add-additional-paths	Whether to make the liveness and readiness health groups available on the main server port.	false
management.endpoint.health.probes.enabled	Whether to enable liveness and readiness probes.	false
management.endpoint.health.roles	Roles used to determine whether a user is authorized to be shown details. When empty, all authenticated users are authorized.	
management.endpoint.health.show-components	When to show components. If not specified the 'show-details' setting will be used.	
management.endpoint.health.show-details	When to show full health details.	never

Name	Description	Default Value
management.endpoint.health.status.http-mapping.*	Mapping of health statuses to HTTP status codes. By default, registered health statuses map to sensible defaults (for example, UP maps to 200).	
management.endpoint.health.status.order	Comma-separated list of health statuses in order of severity.	[DOWN, OUT_OF_SERVICE, UP, UNKNOWN]
management.endpoint.health.validate-group-membership	Whether to validate health group membership on startup. Validation fails if a group includes or excludes a health contributor that does not exist.	true
management.endpoint.heapdump.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.heapdump.enabled	Whether to enable the heapdump endpoint.	true
management.endpoint.httpexchanges.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.httpexchanges.enabled	Whether to enable the httpexchanges endpoint.	true
management.endpoint.info.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.info.enabled	Whether to enable the info endpoint.	true
management.endpoint.integrationgraph.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.integrationgraph.enabled	Whether to enable the integrationgraph endpoint.	true
management.endpoint.liquibase.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.liquibase.enabled	Whether to enable the liquibase endpoint.	true
management.endpoint.logfile.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.logfile.enabled	Whether to enable the logfile endpoint.	true

Name	Description	Default Value
management.endpoint.logfile.external-file	External Logfile to be accessed. Can be used if the logfile is written by output redirect and not by the logging system itself.	
management.endpoint.loggers.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.loggers.enabled	Whether to enable the loggers endpoint.	true
management.endpoint.mappings.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.mappings.enabled	Whether to enable the mappings endpoint.	true
management.endpoint.metrics.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.metrics.enabled	Whether to enable the metrics endpoint.	true
management.endpoint.prometheus.enabled	Whether to enable the prometheus endpoint.	true
management.endpoint.quartz.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.quartz.enabled	Whether to enable the quartz endpoint.	true
management.endpoint.quartz.roles	Roles used to determine whether a user is authorized to be shown unsanitized job or trigger values. When empty, all authenticated users are authorized.	
management.endpoint.quartz.show-values	When to show unsanitized job or trigger values.	
management.endpoint.scheduledtasks.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.scheduledtasks.enabled	Whether to enable the scheduledtasks endpoint.	true
management.endpoint.sessions.enabled	Whether to enable the sessions endpoint.	true
management.endpoint.shutdown.enabled	Whether to enable the shutdown endpoint.	false

Name	Description	Default Value
management.endpoint.startup.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.startup.enabled	Whether to enable the startup endpoint.	true
management.endpoint.threaddump.cache.time-to-live	Maximum time that a response can be cached.	0ms
management.endpoint.threaddump.enabled	Whether to enable the threaddump endpoint.	true
management.endpoints.enabled-by-default	Whether to enable or disable all endpoints by default.	
management.endpoints.jackson.isolated-object-mapper	Whether to use an isolated object mapper to serialize endpoint JSON.	true
management.endpoints.jmx.domain	Endpoints JMX domain name. Fallback to 'spring.jmx.default-domain' if set.	org.springframework.boot
management.endpoints.jmx.exposure.exclude	Endpoint IDs that should be excluded or '*' for all.	
management.endpoints.jmx.exposure.include	Endpoint IDs that should be included or '*' for all.	health
management.endpoints.jmx.static-names	Additional static properties to append to all ObjectNames of MBeans representing Endpoints.	
management.endpoints.migrate-legacy-ids	Whether to transparently migrate legacy endpoint IDs.	false

Name	Description	Default Value
management.endpoints.web.base-path	Base path for Web endpoints. Relative to the servlet context path (server.servlet.context-path) or WebFlux base path (spring.webflux.base-path) when the management server is sharing the main server port. Relative to the management server base path (management.server.base-path) when a separate management server port (management.server.port) is configured.	/actuator
management.endpoints.web.cors.allow-credentials	Whether credentials are supported. When not set, credentials are not supported.	
management.endpoints.web.cors.allowed-headers	Comma-separated list of headers to allow in a request. '*' allows all headers.	
management.endpoints.web.cors.allowed-methods	Comma-separated list of methods to allow. '*' allows all methods. When not set, defaults to GET.	
management.endpoints.web.cors.allowed-origin-patterns	Comma-separated list of origin patterns to allow. Unlike allowed origins which only supports '*', origin patterns are more flexible (for example 'https://*.example.com') and can be used when credentials are allowed. When no allowed origin patterns or allowed origins are set, CORS support is disabled.	

Name	Description	Default Value
management.endpoints.web.cors.allowed-origins	Comma-separated list of origins to allow. '*' allows all origins. When credentials are allowed, '*' cannot be used and origin patterns should be configured instead. When no allowed origins or allowed origin patterns are set, CORS support is disabled.	
management.endpoints.web.cors.exposed-headers	Comma-separated list of headers to include in a response.	
management.endpoints.web.cors.max-age	How long the response from a pre-flight request can be cached by clients. If a duration suffix is not specified, seconds will be used.	1800s
management.endpoints.web.discovery.enabled	Whether the discovery page is enabled.	true
management.endpoints.web.exposure.exclude	Endpoint IDs that should be excluded or '*' for all.	
management.endpoints.web.exposure.include	Endpoint IDs that should be included or '*' for all.	[health]
management.endpoints.web.path-mapping.*	Mapping between endpoint IDs and the path that should expose them.	
management.ganglia.metrics.export.addressing-mode	UDP addressing mode, either unicast or multicast.	multicast
management.ganglia.metrics.export.duration-units	Base time unit used to report durations.	milliseconds
management.ganglia.metrics.export.enabled	Whether exporting of metrics to Ganglia is enabled.	true
management.ganglia.metrics.export.host	Host of the Ganglia server to receive exported metrics.	localhost
management.ganglia.metrics.export.port	Port of the Ganglia server to receive exported metrics.	8649
management.ganglia.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m

Name	Description	Default Value
management.ganglia.metrics.export.time-to-live	Time to live for metrics on Ganglia. Set the multicast Time-To-Live to be one greater than the number of hops (routers) between the hosts.	1
management.graphite.metrics.export.duration-units	Base time unit used to report durations.	milliseconds
management.graphite.metrics.export.enabled	Whether exporting of metrics to Graphite is enabled.	true
management.graphite.metrics.export.graphite-tags-enabled	Whether Graphite tags should be used, as opposed to a hierarchical naming convention. Enabled by default unless "tagsAsPrefix" is set.	
management.graphite.metrics.export.host	Host of the Graphite server to receive exported metrics.	localhost
management.graphite.metrics.export.port	Port of the Graphite server to receive exported metrics.	2004
management.graphite.metrics.export.protocol	Protocol to use while shipping data to Graphite.	pickled
management.graphite.metrics.export.rate-units	Base time unit used to report rates.	seconds
management.graphite.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.graphite.metrics.export.tags-as-prefix	For the hierarchical naming convention, turn the specified tag keys into part of the metric prefix. Ignored if "graphiteTagsEnabled" is true.	[]
management.health.cassandra.enabled	Whether to enable Cassandra health check.	true
management.health.couchbase.enabled	Whether to enable Couchbase health check.	true
management.health.db.enabled	Whether to enable database health check.	true

Name	Description	Default Value
management.health.db.ignore-routing-data-sources	Whether to ignore AbstractRoutingDataSources when creating database health indicators.	false
management.health.defaults.enabled	Whether to enable default health indicators.	true
management.health.diskspace.enabled	Whether to enable disk space health check.	true
management.health.diskspace.path	Path used to compute the available disk space.	
management.health.diskspace.threshold	Minimum disk space that should be available.	10MB
management.health.elasticsearch.enabled	Whether to enable Elasticsearch health check.	true
management.health.influxdb.enabled	Whether to enable InfluxDB health check.	true
management.health.jms.enabled	Whether to enable JMS health check.	true
management.health.ldap.enabled	Whether to enable LDAP health check.	true
management.health.livenessstate.enabled	Whether to enable liveness state health check.	false
management.health.mail.enabled	Whether to enable Mail health check.	true
management.health.mongo.enabled	Whether to enable MongoDB health check.	true
management.health.neo4j.enabled	Whether to enable Neo4j health check.	true
management.health.ping.enabled	Whether to enable ping health check.	true
management.health.rabbit.enabled	Whether to enable RabbitMQ health check.	true
management.health.readinessstate.enabled	Whether to enable readiness state health check.	false
management.health.redis.enabled	Whether to enable Redis health check.	true
management.httpexchanges.recording.enabled	Whether to enable HTTP request-response exchange recording.	true

Name	Description	Default Value
management.httpexchanges.recording.include	Items to be included in the exchange recording. Defaults to request headers (excluding Authorization and Cookie), response headers (excluding Set-Cookie), and time taken.	[request-headers, response-headers, errors]
management.humio.metrics.export.api-token	Humio API token.	
management.humio.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.humio.metrics.export.connect-timeout	Connection timeout for requests to this backend.	5s
management.humio.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.humio.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.humio.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.humio.metrics.export.tags.*	Humio tags describing the data source in which metrics will be stored. Humio tags are a distinct concept from Micrometer's tags. Micrometer's tags are used to divide metrics along dimensional boundaries.	
management.humio.metrics.export.uri	URI to ship metrics to. If you need to publish metrics to an internal proxy en-route to Humio, you can define the location of the proxy with this.	https://cloud.humio.com
management.influx.metrics.export.api-version	API version of InfluxDB to use. Defaults to 'v1' unless an org is configured. If an org is configured, defaults to 'v2'.	

Name	Description	Default Value
management.influx.metrics.export.auto-create-db	Whether to create the Influx database if it does not exist before attempting to publish metrics to it. InfluxDB v1 only.	true
management.influx.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.influx.metrics.export.bucket	Bucket for metrics. Use either the bucket name or ID. Defaults to the value of the db property if not set. InfluxDB v2 only.	
management.influx.metrics.export.compressed	Whether to enable GZIP compression of metrics batches published to Influx.	true
management.influx.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.influx.metrics.export.consistency	Write consistency for each point.	one
management.influx.metrics.export.db	Database to send metrics to. InfluxDB v1 only.	mydb
management.influx.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.influx.metrics.export.org	Org to write metrics to. InfluxDB v2 only.	
management.influx.metrics.export.password	Login password of the Influx server. InfluxDB v1 only.	
management.influx.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.influx.metrics.export.retention-duration	Time period for which Influx should retain data in the current database. For instance 7d, check the influx documentation for more details on the duration format. InfluxDB v1 only.	

Name	Description	Default Value
management.influx.metrics.export.retention-policy	Retention policy to use (Influx writes to the DEFAULT retention policy if one is not specified). InfluxDB v1 only.	
management.influx.metrics.export.retention-replication-factor	How many copies of the data are stored in the cluster. Must be 1 for a single node instance. InfluxDB v1 only.	
management.influx.metrics.export.retention-shard-duration	Time range covered by a shard group. For instance 2w, check the influx documentation for more details on the duration format. InfluxDB v1 only.	
management.influx.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.influx.metrics.export.token	Authentication token to use with calls to the InfluxDB backend. For InfluxDB v1, the Bearer scheme is used. For v2, the Token scheme is used.	
management.influx.metrics.export.uri	URI of the Influx server.	http://localhost:8086
management.influx.metrics.export.user-name	Login user of the Influx server. InfluxDB v1 only.	
management.info.build.enabled	Whether to enable build info.	true
management.info.defaults.enabled	Whether to enable default info contributors.	true
management.info.env.enabled	Whether to enable environment info.	false
management.info.git.enabled	Whether to enable git info.	true
management.info.git.mode	Mode to use to expose git information.	simple
management.info.java.enabled	Whether to enable Java info.	false
management.info.os.enabled	Whether to enable Operating System info.	false
management.jmx.metrics.export.domain	Metrics JMX domain name.	metrics

Name	Description	Default Value
management.jmx.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.jmx.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.kairos.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.kairos.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.kairos.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.kairos.metrics.export.password	Login password of the KairosDB server.	
management.kairos.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.kairos.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.kairos.metrics.export.uri	URI of the KairosDB server.	http://localhost:8080/api/v1/datapoints
management.kairos.metrics.export.user-name	Login user of the KairosDB server.	
management.metrics.data.repository.autotime.enabled	Whether to enable auto-timing.	true
management.metrics.data.repository.autotime.percentiles	Percentiles for which additional time series should be published.	
management.metrics.data.repository.autotime.percentiles-histogram	Whether to publish percentile histograms.	false
management.metrics.data.repository.metric-name	Name of the metric for sent requests.	spring.data.repository.invocations

Name	Description	Default Value
management.metrics.distribution.buffer-length.*	Number of histograms for meter IDs starting with the specified name to keep in the ring buffer. The longest match wins, the key `all` can also be used to configure all meters.	
management.metrics.distribution.expiry.*	Maximum amount of time that samples for meter IDs starting with the specified name are accumulated to decaying distribution statistics before they are reset and rotated. The longest match wins, the key `all` can also be used to configure all meters.	
management.metrics.distribution.maximum-expected-value.*	Maximum value that meter IDs starting with the specified name are expected to observe. The longest match wins. Values can be specified as a double or as a Duration value (for timer meters, defaulting to ms if no unit specified).	
management.metrics.distribution.minimum-expected-value.*	Minimum value that meter IDs starting with the specified name are expected to observe. The longest match wins. Values can be specified as a double or as a Duration value (for timer meters, defaulting to ms if no unit specified).	
management.metrics.distribution.percentiles.*	Specific computed non-aggregable percentiles to ship to the backend for meter IDs starting-with the specified name. The longest match wins, the key 'all' can also be used to configure all meters.	

Name	Description	Default Value
management.metrics.distribution.percentiles-histogram.*	Whether meter IDs starting with the specified name should publish percentile histograms. For monitoring systems that support aggregable percentile calculation based on a histogram, this can be set to true. For other systems, this has no effect. The longest match wins, the key 'all' can also be used to configure all meters.	
management.metrics.distribution.slo.*	Specific service-level objective boundaries for meter IDs starting with the specified name. The longest match wins. Counters will be published for each specified boundary. Values can be specified as a double or as a Duration value (for timer meters, defaulting to ms if no unit specified).	
management.metrics.enable.*	Whether meter IDs starting with the specified name should be enabled. The longest match wins, the key 'all' can also be used to configure all meters.	
management.metrics.mongo.command.enabled	Whether to enable Mongo client command metrics.	true
management.metrics.mongo.connectionpool.enabled	Whether to enable Mongo connection pool metrics.	true
management.metrics.system.diskspace.paths	Comma-separated list of paths to report disk metrics for.	[.]
management.metrics.tags.*	Common tags that are applied to every meter.	

Name	Description	Default Value
management.metrics.use-global-registry	Whether auto-configured MeterRegistry implementations should be bound to the global static registry on Metrics. For testing, set this to 'false' to maximize test independence.	true
management.metrics.web.client.max-uri-tags	Maximum number of unique URI tag values allowed. After the max number of tag values is reached, metrics with additional tag values are denied by filter.	100
management.metrics.web.server.max-uri-tags	Maximum number of unique URI tag values allowed. After the max number of tag values is reached, metrics with additional tag values are denied by filter.	100
management.newrelic.metrics.export.account-id	New Relic account ID.	
management.newrelic.metrics.export.api-key	New Relic API key.	
management.newrelic.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.newrelic.metrics.export.client-provider-type	Client provider type to use.	
management.newrelic.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.newrelic.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true

Name	Description	Default Value
management.newrelic.metrics.export.event-type	The event type that should be published. This property will be ignored if 'meter-name-event-type-enabled' is set to 'true'.	SpringBootSample
management.newrelic.metrics.export.meter-name-event-type-enabled	Whether to send the meter name as the event type instead of using the 'event-type' configuration property value. Can be set to 'true' if New Relic guidelines are not being followed or event types consistent with previous Spring Boot releases are required.	false
management.newrelic.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.newrelic.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.newrelic.metrics.export.uri	URI to ship metrics to.	https://insights-collector.newrelic.com
management.observations.http.client.requests.name	Name of the observation for client requests. If empty, will use the default "http.client.requests".	
management.observations.http.server.requests.name	Name of the observation for server requests. If empty, will use the default "http.server.requests".	
management.otlp.metrics.export.aggregation-temporality	Aggregation temporality of sums. It defines the way additive values are expressed. This setting depends on the backend you use, some only support one temporality.	
management.otlp.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000

Name	Description	Default Value
management.otlp.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.otlp.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.otlp.metrics.export.headers.*	Headers for the exported metrics.	
management.otlp.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.otlp.metrics.export.resource-attributes.*	Monitored resource's attributes.	
management.otlp.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.otlp.metrics.export.url	URI of the OLTP server.	http://localhost:4318/v1/metrics
management.otlp.tracing.compression	Method used to compress the payload.	none
management.otlp.tracing.endpoint	URL to the OTel collector's HTTP API.	http://localhost:4318/v1/traces
management.otlp.tracing.headers.*	Custom HTTP headers you want to pass to the collector, for example auth headers.	
management.otlp.tracing.timeout	Call timeout for the OTel Collector to process an exported batch of data. This timeout spans the entire call: resolving DNS, connecting, writing the request body, server processing, and reading the response body. If the call requires redirects or retries all must complete within one timeout period.	10s
management.prometheus.metrics.export.descriptions	Whether to enable publishing descriptions as part of the scrape payload to Prometheus. Turn this off to minimize the amount of data sent on each scrape.	true

Name	Description	Default Value
management.prometheus.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.prometheus.metrics.export.histogram-flavor	Histogram type for backing DistributionSummary and Timer.	prometheus
management.prometheus.metrics.export.pushgateway.base-url	Base URL for the Pushgateway.	http://localhost:9091
management.prometheus.metrics.export.pushgateway.enabled	Enable publishing over a Prometheus Pushgateway.	false
management.prometheus.metrics.export.pushgateway.grouping-key.*	Grouping key for the pushed metrics.	
management.prometheus.metrics.export.pushgateway.job	Job identifier for this application instance.	
management.prometheus.metrics.export.pushgateway.password	Login password of the Prometheus Pushgateway.	
management.prometheus.metrics.export.pushgateway.push-rate	Frequency with which to push metrics.	1m
management.prometheus.metrics.export.pushgateway.shutdown-operation	Operation that should be performed on shutdown.	none
management.prometheus.metrics.export.pushgateway.username	Login user of the Prometheus Pushgateway.	
management.prometheus.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.server.add-application-context-header	Add the "X-Application-Context" HTTP header in each response.	false
management.server.address	Network address to which the management endpoints should bind. Requires a custom management.server.port.	
management.server.base-path	Management endpoint base path (for instance, '/management'). Requires a custom management.server.port.	

Name	Description	Default Value
management.server.port	Management endpoint HTTP port (uses the same port as the application by default). Configure a different port to use management-specific SSL.	
management.server.ssl.bundle	The name of a configured SSL bundle.	
management.server.ssl.certificate	Path to a PEM-encoded SSL certificate file.	
management.server.ssl.certificate-private-key	Path to a PEM-encoded private key file for the SSL certificate.	
management.server.ssl.ciphers	Supported SSL ciphers.	
management.server.ssl.client-auth	Client authentication mode. Requires a trust store.	
management.server.ssl.enabled	Whether to enable SSL support.	true
management.server.ssl.enabled-protocols	Enabled SSL protocols.	
management.server.ssl.key-alias	Alias that identifies the key in the key store.	
management.server.ssl.key-password	Password used to access the key in the key store.	
management.server.ssl.key-store	Path to the key store that holds the SSL certificate (typically a jks file).	
management.server.ssl.key-store-password	Password used to access the key store.	
management.server.ssl.key-store-provider	Provider for the key store.	
management.server.ssl.key-store-type	Type of the key store.	
management.server.ssl.protocol	SSL protocol to use.	TLS
management.server.ssl.trust-certificate	Path to a PEM-encoded SSL certificate authority file.	
management.server.ssl.trust-certificate-private-key	Path to a PEM-encoded private key file for the SSL certificate authority.	
management.server.ssl.trust-store	Trust store that holds SSL certificates.	

Name	Description	Default Value
management.server.ssl.trust-store-password	Password used to access the trust store.	
management.server.ssl.trust-store-provider	Provider for the trust store.	
management.server.ssl.trust-store-type	Type of the trust store.	
management.signalfx.metrics.export.access-token	SignalFX access token.	
management.signalfx.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.signalfx.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.signalfx.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.signalfx.metrics.export.published-histogram-type	Type of histogram to publish.	default
management.signalfx.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.signalfx.metrics.export.source	Uniquely identifies the app instance that is publishing metrics to SignalFx. Defaults to the local host name.	
management.signalfx.metrics.export.step	Step size (i.e. reporting frequency) to use.	10s
management.signalfx.metrics.export.uri	URI to ship metrics to.	https://ingest.signalfx.com
management.simple.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.simple.metrics.export.mode	Counting mode.	cumulative
management.simple.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m

Name	Description	Default Value
management.stackdriver.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.stackdriver.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.stackdriver.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.stackdriver.metrics.export.metric-type-prefix	Prefix for metric type. Valid prefixes are described in the Google Cloud documentation (https://cloud.google.com/monitoring/custom-metrics#identifier).	custom.googleapis.com/
management.stackdriver.metrics.export.project-id	Identifier of the Google Cloud project to monitor.	
management.stackdriver.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.stackdriver.metrics.export.resource-labels.*	Monitored resource's labels.	
management.stackdriver.metrics.export.resource-type	Monitored resource type.	global
management.stackdriver.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m
management.stackdriver.metrics.export.use-semantic-metric-types	Whether to use semantically correct metric types. When false, counter metrics are published as the GAUGE MetricKind. When true, counter metrics are published as the CUMULATIVE MetricKind.	false
management.statsd.metrics.export.buffered	Whether measurements should be buffered before sending to the StatsD server.	true
management.statsd.metrics.export.enabled	Whether exporting of metrics to StatsD is enabled.	true
management.statsd.metrics.export.flavor	StatsD line protocol to use.	datadog

Name	Description	Default Value
management.statsd.metrics.export.host	Host of the StatsD server to receive exported metrics.	localhost
management.statsd.metrics.export.max-packet-length	Total length of a single payload should be kept within your network's MTU.	1400
management.statsd.metrics.export.polling-frequency	How often gauges will be polled. When a gauge is polled, its value is recalculated and if the value has changed (or publishUnchangedMeters is true), it is sent to the StatsD server.	10s
management.statsd.metrics.export.port	Port of the StatsD server to receive exported metrics.	8125
management.statsd.metrics.export.protocol	Protocol of the StatsD server to receive exported metrics.	udp
management.statsd.metrics.export.publish-unchanged-meters	Whether to send unchanged meters to the StatsD server.	true
management.statsd.metrics.export.step	Step size to use in computing windowed statistics like max. To get the most out of these statistics, align the step interval to be close to your scrape interval.	1m
management.tracing.baggage.correlation.enabled	Whether to enable correlation of the baggage context with logging contexts.	true
management.tracing.baggage.correlation.fields	List of fields that should be correlated with the logging context. That means that these fields would end up as key-value pairs in e.g. MDC.	
management.tracing.baggage.enabled	Whether to enable Micrometer Tracing baggage propagation.	true

Name	Description	Default Value
management.tracing.baggage.remote-fields	List of fields that are referenced the same in-process as it is on the wire. For example, the field "x-vcap-request-id" would be set as-is including the prefix.	
management.tracing.brave.span-joining-supported	Whether the propagation type and tracing backend support sharing the span ID between client and server spans. Requires B3 propagation and a compatible backend.	false
management.tracing.enabled	Whether auto-configuration of tracing is enabled.	true
management.tracing.propagation.consume	Tracing context propagation types consumed by the application.	[W3C, B3, B3_MULTI]
management.tracing.propagation.produce	Tracing context propagation types produced by the application.	[W3C]
management.tracing.propagation.type	Tracing context propagation types produced and consumed by the application. Setting this property overrides the more fine-grained propagation type properties.	
management.tracing.sampling.probability	Probability in the range from 0.0 to 1.0 that a trace will be sampled.	0.1
management.wavefront.api-token	API token used when publishing metrics and traces directly to the Wavefront API host.	
management.wavefront.application.cluster-name	Wavefront Cluster name used in ApplicationTags.	
management.wavefront.application.custom-tags.*	Wavefront custom tags used in ApplicationTags.	
management.wavefront.application.name	Wavefront 'Application' name used in ApplicationTags.	unnamed_application

Name	Description	Default Value
management.wavefront.application.service-name	Wavefront 'Service' name used in ApplicationTags, falling back to 'spring.application.name'. If both are unset it defaults to 'unnamed_service'.	
management.wavefront.application.shard-name	Wavefront Shard name used in ApplicationTags.	
management.wavefront.metrics.export.batch-size	Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.	10000
management.wavefront.metrics.export.connect-timeout	Connection timeout for requests to this backend.	1s
management.wavefront.metrics.export.enabled	Whether exporting of metrics to this backend is enabled.	true
management.wavefront.metrics.export.global-prefix	Global prefix to separate metrics originating from this app's instrumentation from those originating from other Wavefront integrations when viewed in the Wavefront UI.	
management.wavefront.metrics.export.read-timeout	Read timeout for requests to this backend.	10s
management.wavefront.metrics.export.report-day-distribution	Whether to report histogram distributions aggregated into day intervals.	false
management.wavefront.metrics.export.report-hour-distribution	Whether to report histogram distributions aggregated into hour intervals.	false
management.wavefront.metrics.export.report-minute-distribution	Whether to report histogram distributions aggregated into minute intervals.	true
management.wavefront.metrics.export.step	Step size (i.e. reporting frequency) to use.	1m

Name	Description	Default Value
management.wavefront.sender.batch-size	Number of measurements per request to use for Wavefront. If more measurements are found, then multiple requests will be made.	10000
management.wavefront.sender.flush-interval	Flush interval to send queued messages.	1s
management.wavefront.sender.max-queue-size	Maximum size of queued messages.	50000
management.wavefront.sender.message-size	Maximum size of a message.	
management.wavefront.source	Unique identifier for the app instance that is the source of metrics and traces being published to Wavefront. Defaults to the local host name.	
management.wavefront.trace-derived-custom-tag-keys	Customized span tags for RED metrics.	
management.wavefront.uri	URI to ship metrics and traces to.	https://longboard.wavefront.com
management.zipkin.tracing.connect-timeout	Connection timeout for requests to Zipkin.	1s
management.zipkin.tracing.endpoint	URL to the Zipkin API.	http://localhost:9411/api/v2/spans
management.zipkin.tracing.read-timeout	Read timeout for requests to Zipkin.	10s

.A.15. Devtools Properties

Name	Description	Default Value
spring.devtools.add-properties	Whether to enable development property defaults.	true
spring.devtools.livereload.enabled	Whether to enable a livereload.com-compatible server.	true
spring.devtools.livereload.port	Server port.	35729
spring.devtools.remote.context-path	Context path used to handle the remote connection.	/.~~spring-boot!~

Name	Description	Default Value
spring.devtools.remote.proxy.host	The host of the proxy to use to connect to the remote application.	
spring.devtools.remote.proxy.port	The port of the proxy to use to connect to the remote application.	
spring.devtools.remote.restart.enabled	Whether to enable remote restart.	true
spring.devtools.remote.secret	A shared secret required to establish a connection (required to enable remote support).	
spring.devtools.remote.secret-header-name	HTTP header used to transfer the shared secret.	X-AUTH-TOKEN
spring.devtools.restart.additional-exclude	Additional patterns that should be excluded from triggering a full restart.	
spring.devtools.restart.additional-paths	Additional paths to watch for changes.	
spring.devtools.restart.enabled	Whether to enable automatic restart.	true
spring.devtools.restart.exclude	Patterns that should be excluded from triggering a full restart.	META-INF/maven/**,META-INF/resources/**,resources/**,static/**,public/**,templates/**,**/*Test.class,**/*Tests.class,git.properties,META-INF/build-info.properties
spring.devtools.restart.log-condition-evaluation-delta	Whether to log the condition evaluation delta upon restart.	true
spring.devtools.restart.poll-interval	Amount of time to wait between polling for classpath changes.	1s
spring.devtools.restart.quiet-period	Amount of quiet time required without any classpath changes before a restart is triggered.	400ms

Name	Description	Default Value
spring.devtools.restart.trigger-file	Name of a specific file that, when changed, triggers the restart check. Must be a simple name (without any path) of a file that appears on your classpath. If not specified, any classpath file change triggers the restart.	

.A.16. Docker Compose Properties

Name	Description	Default Value
spring.docker.compose.enabled	Whether docker compose support is enabled.	true
spring.docker.compose.file	Path to a specific docker compose configuration file.	
spring.docker.compose.host	Hostname or IP of the machine where the docker containers are started.	
spring.docker.compose.lifecycle-management	Docker compose lifecycle management.	start-and-stop
spring.docker.compose.profiles.active	Docker compose profiles that should be active.	
spring.docker.compose.readiness.tcp.connect-timeout	Timeout for connections.	200ms
spring.docker.compose.readiness.tcp.read-timeout	Timeout for reads.	200ms
spring.docker.compose.readiness.timeout	Timeout of the readiness checks.	2m
spring.docker.compose.readiness.wait	Wait strategy to use.	always
spring.docker.compose.skip.in-tests	Whether to skip in tests.	true
spring.docker.compose.start.command	Command used to start docker compose.	up
spring.docker.compose.start.log-level	Log level for output.	info
spring.docker.compose.stop.command	Command used to stop docker compose.	stop
spring.docker.compose.stop.timeout	Timeout for stopping Docker Compose. Use '0' for forced stop.	10s

.A.17. Testing Properties

Name	Description	Default Value
`spring.test.database.replace`	Type of existing DataSource to replace.	`any`
`spring.test.mockmvc.print`	MVC Print option.	`default`
`spring.test.observability.auto-configure`	Whether observability should be auto-configured in tests.	`false`

Appendix B: Configuration Metadata

Spring Boot jars include metadata files that provide details of all supported configuration properties. The files are designed to let IDE developers offer contextual help and “code completion” as users are working with `application.properties` or `application.yaml` files.

The majority of the metadata file is generated automatically at compile time by processing all items annotated with `@ConfigurationProperties`. However, it is possible to write part of the metadata manually for corner cases or more advanced use cases.

.B.1. Metadata Format

Configuration metadata files are located inside jars under `META-INF/spring-configuration-metadata.json`. They use a JSON format with items categorized under either “groups” or “properties” and additional values hints categorized under "hints", as shown in the following example:

```
{"groups": [
    {
        "name": "server",
        "type": "org.springframework.boot.autoconfigure.web.ServerProperties",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
        "name": "spring.jpa.hibernate",
        "type":
"org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate",
        "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties",
        "sourceMethod": "getHibernate()"
    }
    ...
],"properties": [
    {
        "name": "server.port",
        "type": "java.lang.Integer",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
```

```
        "name": "server.address",
        "type": "java.net.InetAddress",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
          "name": "spring.jpa.hibernate.ddl-auto",
          "type": "java.lang.String",
          "description": "DDL mode. This is actually a shortcut for the
\"hibernate.hbm2ddl.auto\" property.",
          "sourceType":
"org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"
    }
    ...
],"hints": [
    {
        "name": "spring.jpa.hibernate.ddl-auto",
        "values": [
            {
                "value": "none",
                "description": "Disable DDL handling."
            },
            {
                "value": "validate",
                "description": "Validate the schema, make no changes to the database."
            },
            {
                "value": "update",
                "description": "Update the schema if necessary."
            },
            {
                "value": "create",
                "description": "Create the schema and destroy previous data."
            },
            {
                "value": "create-drop",
                "description": "Create and then destroy the schema at the end of the
session."
            }
        ]
    }
]}
```

Each “property” is a configuration item that the user specifies with a given value. For example, `server.port` and `server.address` might be specified in your `application.properties`/`application.yaml`, as follows:

Properties

```
server.port=9090
server.address=127.0.0.1
```

Yaml

```
server:
  port: 9090
  address: 127.0.0.1
```

The “groups” are higher level items that do not themselves specify a value but instead provide a contextual grouping for properties. For example, the `server.port` and `server.address` properties are part of the `server` group.

NOTE	It is not required that every “property” has a “group”. Some properties might exist in their own right.

Finally, “hints” are additional information used to assist the user in configuring a given property. For example, when a developer is configuring the `spring.jpa.hibernate.ddl-auto` property, a tool can use the hints to offer some auto-completion help for the `none`, `validate`, `update`, `create`, and `create-drop` values.

Group Attributes

The JSON object contained in the `groups` array can contain the attributes shown in the following table:

Name	Type	Purpose
`name`	String	The full name of the group. This attribute is mandatory.
`type`	String	The class name of the data type of the group. For example, if the group were based on a class annotated with `@ConfigurationProperties`, the attribute would contain the fully qualified name of that class. If it were based on a `@Bean` method, it would be the return type of that method. If the type is not known, the attribute may be omitted.
`description`	String	A short description of the group that can be displayed to users. If no description is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`sourceType`	String	The class name of the source that contributed this group. For example, if the group were based on a `@Bean` method annotated with `@ConfigurationProperties`, this attribute would contain the fully qualified name of the `@Configuration` class that contains the method. If the source type is not known, the attribute may be omitted.
`sourceMethod`	String	The full name of the method (include parenthesis and argument types) that contributed this group (for example, the name of a `@ConfigurationProperties` annotated `@Bean` method). If the source method is not known, it may be omitted.

Property Attributes

The JSON object contained in the `properties` array can contain the attributes described in the following table:

Name	Type	Purpose
`name`	String	The full name of the property. Names are in lower-case period-separated form (for example, `server.address`). This attribute is mandatory.
`type`	String	The full signature of the data type of the property (for example, `java.lang.String`) but also a full generic type (such as `java.util.Map<java.lang.String,com.example.MyEnum>`). You can use this attribute to guide the user as to the types of values that they can enter. For consistency, the type of a primitive is specified by using its wrapper counterpart (for example, `boolean` becomes `java.lang.Boolean`). Note that this class may be a complex type that gets converted from a `String` as values are bound. If the type is not known, it may be omitted.
`description`	String	A short description of the property that can be displayed to users. If no description is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`sourceType`	String	The class name of the source that contributed this property. For example, if the property were from a class annotated with `@ConfigurationProperties`, this attribute would contain the fully qualified name of that class. If the source type is unknown, it may be omitted.
`defaultValue`	Object	The default value, which is used if the property is not specified. If the type of the property is an array, it can be an array of value(s). If the default value is unknown, it may be omitted.
`deprecation`	Deprecation	Specify whether the property is deprecated. If the field is not deprecated or if that information is not known, it may be omitted. The next table offers more detail about the `deprecation` attribute.

The JSON object contained in the `deprecation` attribute of each `properties` element can contain the following attributes:

Name	Type	Purpose
`level`	String	The level of deprecation, which can be either `warning` (the default) or `error`. When a property has a `warning` deprecation level, it should still be bound in the environment. However, when it has an `error` deprecation level, the property is no longer managed and is not bound.

Name	Type	Purpose
`reason`	String	A short description of the reason why the property was deprecated. If no reason is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`replacement`	String	The full name of the property that *replaces* this deprecated property. If there is no replacement for this property, it may be omitted.

NOTE

Prior to Spring Boot 1.3, a single `deprecated` boolean attribute can be used instead of the `deprecation` element. This is still supported in a deprecated fashion and should no longer be used. If no reason and replacement are available, an empty `deprecation` object should be set.

Deprecation can also be specified declaratively in code by adding the `@DeprecatedConfigurationProperty` annotation to the getter exposing the deprecated property. For instance, assume that the `my.app.target` property was confusing and was renamed to `my.app.name`. The following example shows how to handle that situation:

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.DeprecatedConfigurationProperty;

@ConfigurationProperties("my.app")
public class MyProperties {

    private String name;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Deprecated
    @DeprecatedConfigurationProperty(replacement = "my.app.name")
    public String getTarget() {
        return this.name;
    }

    @Deprecated
    public void setTarget(String target) {
        this.name = target;
    }

}
```

NOTE	There is no way to set a `level`. `warning` is always assumed, since code is still handling the property.

The preceding code makes sure that the deprecated property still works (delegating to the `name` property behind the scenes). Once the `getTarget` and `setTarget` methods can be removed from your public API, the automatic deprecation hint in the metadata goes away as well. If you want to keep a hint, adding manual metadata with an `error` deprecation level ensures that users are still informed about that property. Doing so is particularly useful when a `replacement` is provided.

Hint Attributes

The JSON object contained in the `hints` array can contain the attributes shown in the following table:

Name	Type	Purpose
`name`	String	The full name of the property to which this hint refers. Names are in lower-case period-separated form (such as `spring.mvc.servlet.path`). If the property refers to a map (such as `system.contexts`), the hint either applies to the *keys* of the map (`system.contexts.keys`) or the *values* (`system.contexts.values`) of the map. This attribute is mandatory.
`values`	ValueHint[]	A list of valid values as defined by the `ValueHint` object (described in the next table). Each entry defines the value and may have a description.
`providers`	ValueProvider[]	A list of providers as defined by the `ValueProvider` object (described later in this document). Each entry defines the name of the provider and its parameters, if any.

The JSON object contained in the `values` attribute of each `hint` element can contain the attributes described in the following table:

Name	Type	Purpose
`value`	Object	A valid value for the element to which the hint refers. If the type of the property is an array, it can also be an array of value(s). This attribute is mandatory.
`description`	String	A short description of the value that can be displayed to users. If no description is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).

The JSON object contained in the `providers` attribute of each `hint` element can contain the attributes described in the following table:

Name	Type	Purpose
`name`	String	The name of the provider to use to offer additional content assistance for the element to which the hint refers.
`parameters`	JSON object	Any additional parameter that the provider supports (check the documentation of the provider for more details).

Repeated Metadata Items

Objects with the same “property” and “group” name can appear multiple times within a metadata file. For example, you could bind two separate classes to the same prefix, with each having potentially overlapping property names. While the same names appearing in the metadata multiple times should not be common, consumers of metadata should take care to ensure that they support it.

.B.2. Providing Manual Hints

To improve the user experience and further assist the user in configuring a given property, you can provide additional metadata that:

- Describes the list of potential values for a property.
- Associates a provider, to attach a well defined semantic to a property, so that a tool can discover the list of potential values based on the project's context.

Value Hint

The `name` attribute of each hint refers to the `name` of a property. In the initial example shown earlier, we provide five values for the `spring.jpa.hibernate.ddl-auto` property: `none`, `validate`, `update`, `create`, and `create-drop`. Each value may have a description as well.

If your property is of type `Map`, you can provide hints for both the keys and the values (but not for the map itself). The special `.keys` and `.values` suffixes must refer to the keys and the values, respectively.

Assume a `my.contexts` maps magic `String` values to an integer, as shown in the following example:

```
import java.util.Map;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("my")
public class MyProperties {

    private Map<String, Integer> contexts;

    public Map<String, Integer> getContexts() {
        return this.contexts;
    }

    public void setContexts(Map<String, Integer> contexts) {
        this.contexts = contexts;
    }

}
```

The magic values are (in this example) are `sample1` and `sample2`. In order to offer additional content assistance for the keys, you could add the following JSON to the manual metadata of the module:

```
{"hints": [
    {
        "name": "my.contexts.keys",
        "values": [
            {
                "value": "sample1"
            },
            {
                "value": "sample2"
            }
        ]
    }
]}
```

TIP We recommend that you use an `Enum` for those two values instead. If your IDE supports it, this is by far the most effective approach to auto-completion.

Value Providers

Providers are a powerful way to attach semantics to a property. In this section, we define the official providers that you can use for your own hints. However, your favorite IDE may implement some of these or none of them. Also, it could eventually provide its own.

NOTE As this is a new feature, IDE vendors must catch up with how it works. Adoption times naturally vary.

The following table summarizes the list of supported providers:

Name	Description
`any`	Permits any additional value to be provided.
`class-reference`	Auto-completes the classes available in the project. Usually constrained by a base class that is specified by the `target` parameter.
`handle-as`	Handles the property as if it were defined by the type defined by the mandatory `target` parameter.
`logger-name`	Auto-completes valid logger names and logger groups. Typically, package and class names available in the current project can be auto-completed as well as defined groups.
`spring-bean-reference`	Auto-completes the available bean names in the current project. Usually constrained by a base class that is specified by the `target` parameter.
`spring-profile-name`	Auto-completes the available Spring profile names in the project.

TIP	Only one provider can be active for a given property, but you can specify several providers if they can all manage the property *in some way*. Make sure to place the most powerful provider first, as the IDE must use the first one in the JSON section that it can handle. If no provider for a given property is supported, no special content assistance is provided, either.

Any

The special **any** provider value permits any additional values to be provided. Regular value validation based on the property type should be applied if this is supported.

This provider is typically used if you have a list of values and any extra values should still be considered as valid.

The following example offers `on` and `off` as auto-completion values for `system.state`:

```
{"hints": [
    {
        "name": "system.state",
        "values": [
            {
                "value": "on"
            },
            {
                "value": "off"
            }
        ],
        "providers": [
            {
                "name": "any"
            }
        ]
    }
]}
```

Note that, in the preceding example, any other value is also allowed.

Class Reference

The **class-reference** provider auto-completes classes available in the project. This provider supports the following parameters:

Parameter	Type	Default value	Description
`target`	`String` (`Class`)	*none*	The fully qualified name of the class that should be assignable to the chosen value. Typically used to filter out-non candidate classes. Note that this information can be provided by the type itself by exposing a class with the appropriate upper bound.
`concrete`	`boolean`	true	Specify whether only concrete classes are to be considered as valid candidates.

The following metadata snippet corresponds to the standard `server.servlet.jsp.class-name` property that defines the `JspServlet` class name to use:

```
{"hints": [
    {
        "name": "server.servlet.jsp.class-name",
        "providers": [
            {
                "name": "class-reference",
                "parameters": {
                    "target": "jakarta.servlet.http.HttpServlet"
                }
            }
        ]
    }
]}
```

Handle As

The **handle-as** provider lets you substitute the type of the property to a more high-level type. This typically happens when the property has a `java.lang.String` type, because you do not want your configuration classes to rely on classes that may not be on the classpath. This provider supports the following parameters:

Parameter	Type	Default value	Description
`target`	`String` (`Class`)	*none*	The fully qualified name of the type to consider for the property. This parameter is mandatory.

The following types can be used:

- Any `java.lang.Enum`: Lists the possible values for the property. (We recommend defining the property with the `Enum` type, as no further hint should be required for the IDE to auto-complete the values)
- `java.nio.charset.Charset`: Supports auto-completion of charset/encoding values (such as `UTF-8`)
- `java.util.Locale`: auto-completion of locales (such as `en_US`)
- `org.springframework.util.MimeType`: Supports auto-completion of content type values (such as

`text/plain`)

- `org.springframework.core.io.Resource`: Supports auto-completion of Spring's Resource abstraction to refer to a file on the filesystem or on the classpath (such as `classpath:/sample.properties`)

TIP	If multiple values can be provided, use a `Collection` or *Array* type to teach the IDE about it.

The following metadata snippet corresponds to the standard `spring.liquibase.change-log` property that defines the path to the changelog to use. It is actually used internally as a `org.springframework.core.io.Resource` but cannot be exposed as such, because we need to keep the original String value to pass it to the Liquibase API.

```
{"hints": [
    {
        "name": "spring.liquibase.change-log",
        "providers": [
            {
                "name": "handle-as",
                "parameters": {
                    "target": "org.springframework.core.io.Resource"
                }
            }
        ]
    }
]}
```

Logger Name

The **logger-name** provider auto-completes valid logger names and logger groups. Typically, package and class names available in the current project can be auto-completed. If groups are enabled (default) and if a custom logger group is identified in the configuration, auto-completion for it should be provided. Specific frameworks may have extra magic logger names that can be supported as well.

This provider supports the following parameters:

Parameter	Type	Default value	Description
`group`	`boolean`	`true`	Specify whether known groups should be considered.

Since a logger name can be any arbitrary name, this provider should allow any value but could highlight valid package and class names that are not available in the project's classpath.

The following metadata snippet corresponds to the standard `logging.level` property. Keys are *logger names*, and values correspond to the standard log levels or any custom level. As Spring Boot defines a few logger groups out-of-the-box, dedicated value hints have been added for those.

```
{"hints": [
    {
        "name": "logging.level.keys",
        "values": [
            {
                "value": "root",
                "description": "Root logger used to assign the default logging level."
            },
            {
                "value": "sql",
                "description": "SQL logging group including Hibernate SQL logger."
            },
            {
                "value": "web",
                "description": "Web logging group including codecs."
            }
        ],
        "providers": [
            {
                "name": "logger-name"
            }
        ]
    },
    {
        "name": "logging.level.values",
        "values": [
            {
                "value": "trace"
            },
            {
                "value": "debug"
            },
            {
                "value": "info"
            },
            {
                "value": "warn"
            },
            {
                "value": "error"
            },
            {
                "value": "fatal"
            },
            {
                "value": "off"
            }

        ],
        "providers": [
            {
```

```
                "name": "any"
            }
        ]
    }
]}
```

Spring Bean Reference

The **spring-bean-reference** provider auto-completes the beans that are defined in the configuration of the current project. This provider supports the following parameters:

Parameter	Type	Default value	Description
`target`	`String` (`Class`)	*none*	The fully qualified name of the bean class that should be assignable to the candidate. Typically used to filter out non-candidate beans.

The following metadata snippet corresponds to the standard `spring.jmx.server` property that defines the name of the `MBeanServer` bean to use:

```
{"hints": [
    {
        "name": "spring.jmx.server",
        "providers": [
            {
                "name": "spring-bean-reference",
                "parameters": {
                    "target": "javax.management.MBeanServer"
                }
            }
        ]
    }
]}
```

NOTE	The binder is not aware of the metadata. If you provide that hint, you still need to transform the bean name into an actual Bean reference using by the `ApplicationContext`.

Spring Profile Name

The **spring-profile-name** provider auto-completes the Spring profiles that are defined in the configuration of the current project.

The following metadata snippet corresponds to the standard `spring.profiles.active` property that defines the name of the Spring profile(s) to enable:

```
{"hints": [
    {
        "name": "spring.profiles.active",
        "providers": [
            {
                "name": "spring-profile-name"
            }
        ]
    }
]}
```

.B.3. Generating Your Own Metadata by Using the Annotation Processor

You can easily generate your own configuration metadata file from items annotated with `@ConfigurationProperties` by using the `spring-boot-configuration-processor` jar. The jar includes a Java annotation processor which is invoked as your project is compiled.

Configuring the Annotation Processor

To use the processor, include a dependency on `spring-boot-configuration-processor`.

With Maven the dependency should be declared as optional, as shown in the following example:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

With Gradle, the dependency should be declared in the `annotationProcessor` configuration, as shown in the following example:

```
dependencies {
    annotationProcessor "org.springframework.boot:spring-boot-configuration-processor"
}
```

If you are using an `additional-spring-configuration-metadata.json` file, the `compileJava` task should be configured to depend on the `processResources` task, as shown in the following example:

```
tasks.named('compileJava') {
    inputs.files(tasks.named('processResources'))
}
```

This dependency ensures that the additional metadata is available when the annotation processor runs during compilation.

If you are using AspectJ in your project, you need to make sure that the annotation processor runs only once. There are several ways to do this. With Maven, you can configure the `maven-apt-plugin` explicitly and add the dependency to the annotation processor only there. You could also let the AspectJ plugin run all the processing and disable annotation processing in the `maven-compiler-plugin` configuration, as follows:

NOTE

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <proc>none</proc>
    </configuration>
</plugin>
```

NOTE

If you are using Lombok in your project, you need to make sure that its annotation processor runs before `spring-boot-configuration-processor`. To do so with Maven, you can list the annotation processors in the right order using the `annotationProcessors` attribute of the Maven compiler plugin. If you are not using this attribute, and annotation processors are picked up by the dependencies available on the classpath, make sure that the `lombok` dependency is defined before the `spring-boot-configuration-processor` dependency.

Automatic Metadata Generation

The processor picks up both classes and methods that are annotated with `@ConfigurationProperties`.

If the class has a single parameterized constructor, one property is created per constructor parameter, unless the constructor is annotated with `@Autowired`. If the class has a constructor explicitly annotated with `@ConstructorBinding`, one property is created per constructor parameter for that constructor. Otherwise, properties are discovered through the presence of standard getters and setters with special handling for collection and map types (that is detected even if only a getter is present). The annotation processor also supports the use of the `@Data`, `@Value`, `@Getter`, and `@Setter` lombok annotations.

Consider the following example:

```
import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "my.server")
public class MyServerProperties {

    /**
     * Name of the server.
     */
    private String name;

    /**
     * IP address to listen to.
     */
    private String ip = "127.0.0.1";

    /**
     * Port to listener to.
     */
    private int port = 9797;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getIp() {
        return this.ip;
    }

    public void setIp(String ip) {
        this.ip = ip;
    }

    public int getPort() {
        return this.port;
    }

    public void setPort(int port) {
        this.port = port;
    }
    // fold:off

}
```

This exposes three properties where `my.server.name` has no default and `my.server.ip` and `my.server.port` defaults to `"127.0.0.1"` and `9797` respectively. The Javadoc on fields is used to

populate the `description` attribute. For instance, the description of `my.server.ip` is "IP address to listen to.".

NOTE	You should only use plain text with `@ConfigurationProperties` field Javadoc, since they are not processed before being added to the JSON.

The annotation processor applies a number of heuristics to extract the default value from the source model. Default values have to be provided statically. In particular, do not refer to a constant defined in another class. Also, the annotation processor cannot auto-detect default values for `Enum`s and `Collections`s.

For cases where the default value could not be detected, manual metadata should be provided. Consider the following example:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "my.messaging")
public class MyMessagingProperties {

    private List<String> addresses = new ArrayList<>(Arrays.asList("a", "b"));

    private ContainerType containerType = ContainerType.SIMPLE;

    public List<String> getAddresses() {
        return this.addresses;
    }

    public void setAddresses(List<String> addresses) {
        this.addresses = addresses;
    }

    public ContainerType getContainerType() {
        return this.containerType;
    }

    public void setContainerType(ContainerType containerType) {
        this.containerType = containerType;
    }

    public enum ContainerType {

        SIMPLE, DIRECT

    }

}
```

In order to document default values for properties in the class above, you could add the following content to the manual metadata of the module:

```
{"properties": [
    {
        "name": "my.messaging.addresses",
        "defaultValue": ["a", "b"]
    },
    {
        "name": "my.messaging.container-type",
        "defaultValue": "simple"
    }
]}
```

NOTE	Only the `name` of the property is required to document additional metadata for existing properties.

Nested Properties

The annotation processor automatically considers inner classes as nested properties. Rather than documenting the `ip` and `port` at the root of the namespace, we could create a sub-namespace for it. Consider the updated example:

```
import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "my.server")
public class MyServerProperties {

    private String name;

    private Host host;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Host getHost() {
        return this.host;
    }

    public void setHost(Host host) {
        this.host = host;
    }

    public static class Host {

        private String ip;

        private int port;

        public String getIp() {
            return this.ip;
        }

        public void setIp(String ip) {
            this.ip = ip;
        }

        public int getPort() {
            return this.port;
        }

        public void setPort(int port) {
            this.port = port;
        }

    }

}
```

The preceding example produces metadata information for `my.server.name`, `my.server.host.ip`, and `my.server.host.port` properties. You can use the `@NestedConfigurationProperty` annotation on a field to indicate that a regular (non-inner) class should be treated as if it were nested.

TIP	This has no effect on collections and maps, as those types are automatically identified, and a single metadata property is generated for each of them.

Adding Additional Metadata

Spring Boot's configuration file handling is quite flexible, and it is often the case that properties may exist that are not bound to a `@ConfigurationProperties` bean. You may also need to tune some attributes of an existing key. To support such cases and let you provide custom "hints", the annotation processor automatically merges items from `META-INF/additional-spring-configuration-metadata.json` into the main metadata file.

If you refer to a property that has been detected automatically, the description, default value, and deprecation information are overridden, if specified. If the manual property declaration is not identified in the current module, it is added as a new property.

The format of the `additional-spring-configuration-metadata.json` file is exactly the same as the regular `spring-configuration-metadata.json`. The additional properties file is optional. If you do not have any additional properties, do not add the file.

Appendix C: Auto-configuration Classes

This appendix contains details of all of the auto-configuration classes provided by Spring Boot, with links to documentation and source code. Remember to also look at the conditions report in your application for more details of which features are switched on. (To do so, start the app with `--debug` or `-Ddebug` or, in an Actuator application, use the `conditions` endpoint).

.C.1. spring-boot-autoconfigure

The following auto-configuration classes are from the `spring-boot-autoconfigure` module:

Configuration Class	Links
`ActiveMQAutoConfiguration`	javadoc
`AopAutoConfiguration`	javadoc
`ApplicationAvailabilityAutoConfiguration`	javadoc
`ArtemisAutoConfiguration`	javadoc
`BatchAutoConfiguration`	javadoc
`CacheAutoConfiguration`	javadoc
`CassandraAutoConfiguration`	javadoc
`CassandraDataAutoConfiguration`	javadoc
`CassandraReactiveDataAutoConfiguration`	javadoc

Configuration Class	Links
CassandraReactiveRepositoriesAutoConfiguration	javadoc
CassandraRepositoriesAutoConfiguration	javadoc
ClientHttpConnectorAutoConfiguration	javadoc
CodecsAutoConfiguration	javadoc
ConfigurationPropertiesAutoConfiguration	javadoc
CouchbaseAutoConfiguration	javadoc
CouchbaseDataAutoConfiguration	javadoc
CouchbaseReactiveDataAutoConfiguration	javadoc
CouchbaseReactiveRepositoriesAutoConfiguration	javadoc
CouchbaseRepositoriesAutoConfiguration	javadoc
DataSourceAutoConfiguration	javadoc
DataSourceTransactionManagerAutoConfiguration	javadoc
DispatcherServletAutoConfiguration	javadoc
ElasticsearchClientAutoConfiguration	javadoc
ElasticsearchDataAutoConfiguration	javadoc
ElasticsearchRepositoriesAutoConfiguration	javadoc
ElasticsearchRestClientAutoConfiguration	javadoc
EmbeddedLdapAutoConfiguration	javadoc
EmbeddedWebServerFactoryCustomizerAutoConfiguration	javadoc
ErrorMvcAutoConfiguration	javadoc
ErrorWebFluxAutoConfiguration	javadoc
FlywayAutoConfiguration	javadoc
FreeMarkerAutoConfiguration	javadoc
GraphQlAutoConfiguration	javadoc
GraphQlQueryByExampleAutoConfiguration	javadoc
GraphQlQuerydslAutoConfiguration	javadoc
GraphQlRSocketAutoConfiguration	javadoc
GraphQlReactiveQueryByExampleAutoConfiguration	javadoc
GraphQlReactiveQuerydslAutoConfiguration	javadoc
GraphQlWebFluxAutoConfiguration	javadoc
GraphQlWebFluxSecurityAutoConfiguration	javadoc
GraphQlWebMvcAutoConfiguration	javadoc
GraphQlWebMvcSecurityAutoConfiguration	javadoc

Configuration Class	Links
GroovyTemplateAutoConfiguration	javadoc
GsonAutoConfiguration	javadoc
H2ConsoleAutoConfiguration	javadoc
HazelcastAutoConfiguration	javadoc
HazelcastJpaDependencyAutoConfiguration	javadoc
HibernateJpaAutoConfiguration	javadoc
HttpEncodingAutoConfiguration	javadoc
HttpHandlerAutoConfiguration	javadoc
HttpMessageConvertersAutoConfiguration	javadoc
HypermediaAutoConfiguration	javadoc
InfluxDbAutoConfiguration	javadoc
IntegrationAutoConfiguration	javadoc
JacksonAutoConfiguration	javadoc
JdbcRepositoriesAutoConfiguration	javadoc
JdbcTemplateAutoConfiguration	javadoc
JerseyAutoConfiguration	javadoc
JmsAutoConfiguration	javadoc
JmxAutoConfiguration	javadoc
JndiConnectionFactoryAutoConfiguration	javadoc
JndiDataSourceAutoConfiguration	javadoc
JooqAutoConfiguration	javadoc
JpaRepositoriesAutoConfiguration	javadoc
JsonbAutoConfiguration	javadoc
JtaAutoConfiguration	javadoc
KafkaAutoConfiguration	javadoc
LdapAutoConfiguration	javadoc
LdapRepositoriesAutoConfiguration	javadoc
LifecycleAutoConfiguration	javadoc
LiquibaseAutoConfiguration	javadoc
MailSenderAutoConfiguration	javadoc
MailSenderValidatorAutoConfiguration	javadoc
MessageSourceAutoConfiguration	javadoc
MongoAutoConfiguration	javadoc

Configuration Class	Links
MongoDataAutoConfiguration	javadoc
MongoReactiveAutoConfiguration	javadoc
MongoReactiveDataAutoConfiguration	javadoc
MongoReactiveRepositoriesAutoConfiguration	javadoc
MongoRepositoriesAutoConfiguration	javadoc
MultipartAutoConfiguration	javadoc
MustacheAutoConfiguration	javadoc
Neo4jAutoConfiguration	javadoc
Neo4jDataAutoConfiguration	javadoc
Neo4jReactiveDataAutoConfiguration	javadoc
Neo4jReactiveRepositoriesAutoConfiguration	javadoc
Neo4jRepositoriesAutoConfiguration	javadoc
NettyAutoConfiguration	javadoc
OAuth2AuthorizationServerAutoConfiguration	javadoc
OAuth2AuthorizationServerJwtAutoConfiguration	javadoc
OAuth2ClientAutoConfiguration	javadoc
OAuth2ResourceServerAutoConfiguration	javadoc
PersistenceExceptionTranslationAutoConfiguration	javadoc
ProjectInfoAutoConfiguration	javadoc
PropertyPlaceholderAutoConfiguration	javadoc
QuartzAutoConfiguration	javadoc
R2dbcAutoConfiguration	javadoc
R2dbcDataAutoConfiguration	javadoc
R2dbcRepositoriesAutoConfiguration	javadoc
R2dbcTransactionManagerAutoConfiguration	javadoc
RSocketGraphQlClientAutoConfiguration	javadoc
RSocketMessagingAutoConfiguration	javadoc
RSocketRequesterAutoConfiguration	javadoc
RSocketSecurityAutoConfiguration	javadoc
RSocketServerAutoConfiguration	javadoc
RSocketStrategiesAutoConfiguration	javadoc
RabbitAutoConfiguration	javadoc
ReactiveElasticsearchClientAutoConfiguration	javadoc

Configuration Class	Links
ReactiveElasticsearchRepositoriesAutoConfiguration	javadoc
ReactiveMultipartAutoConfiguration	javadoc
ReactiveOAuth2ClientAutoConfiguration	javadoc
ReactiveOAuth2ResourceServerAutoConfiguration	javadoc
ReactiveSecurityAutoConfiguration	javadoc
ReactiveUserDetailsServiceAutoConfiguration	javadoc
ReactiveWebServerFactoryAutoConfiguration	javadoc
RedisAutoConfiguration	javadoc
RedisReactiveAutoConfiguration	javadoc
RedisRepositoriesAutoConfiguration	javadoc
RepositoryRestMvcAutoConfiguration	javadoc
RestTemplateAutoConfiguration	javadoc
Saml2RelyingPartyAutoConfiguration	javadoc
SecurityAutoConfiguration	javadoc
SecurityFilterAutoConfiguration	javadoc
SendGridAutoConfiguration	javadoc
ServletWebServerFactoryAutoConfiguration	javadoc
SessionAutoConfiguration	javadoc
SpringApplicationAdminJmxAutoConfiguration	javadoc
SpringDataWebAutoConfiguration	javadoc
SqlInitializationAutoConfiguration	javadoc
SslAutoConfiguration	javadoc
TaskExecutionAutoConfiguration	javadoc
TaskSchedulingAutoConfiguration	javadoc
ThymeleafAutoConfiguration	javadoc
TransactionAutoConfiguration	javadoc
UserDetailsServiceAutoConfiguration	javadoc
ValidationAutoConfiguration	javadoc
WebClientAutoConfiguration	javadoc
WebFluxAutoConfiguration	javadoc
WebMvcAutoConfiguration	javadoc
WebServiceTemplateAutoConfiguration	javadoc
WebServicesAutoConfiguration	javadoc

Configuration Class	Links
WebSessionIdResolverAutoConfiguration	javadoc
WebSocketMessagingAutoConfiguration	javadoc
WebSocketReactiveAutoConfiguration	javadoc
WebSocketServletAutoConfiguration	javadoc
XADataSourceAutoConfiguration	javadoc

.C.2. spring-boot-actuator-autoconfigure

The following auto-configuration classes are from the `spring-boot-actuator-autoconfigure` module:

Configuration Class	Links
AppOpticsMetricsExportAutoConfiguration	javadoc
AtlasMetricsExportAutoConfiguration	javadoc
AuditAutoConfiguration	javadoc
AuditEventsEndpointAutoConfiguration	javadoc
AvailabilityHealthContributorAutoConfiguration	javadoc
AvailabilityProbesAutoConfiguration	javadoc
BatchObservationAutoConfiguration	javadoc
BeansEndpointAutoConfiguration	javadoc
BraveAutoConfiguration	javadoc
CacheMetricsAutoConfiguration	javadoc
CachesEndpointAutoConfiguration	javadoc
CassandraHealthContributorAutoConfiguration	javadoc
CassandraReactiveHealthContributorAutoConfiguration	javadoc
CloudFoundryActuatorAutoConfiguration	javadoc
CompositeMeterRegistryAutoConfiguration	javadoc
ConditionsReportEndpointAutoConfiguration	javadoc
ConfigurationPropertiesReportEndpointAutoConfiguration	javadoc
ConnectionFactoryHealthContributorAutoConfiguration	javadoc
ConnectionPoolMetricsAutoConfiguration	javadoc
CouchbaseHealthContributorAutoConfiguration	javadoc
CouchbaseReactiveHealthContributorAutoConfiguration	javadoc
DataSourceHealthContributorAutoConfiguration	javadoc
DataSourcePoolMetricsAutoConfiguration	javadoc
DatadogMetricsExportAutoConfiguration	javadoc

Configuration Class	Links
DiskSpaceHealthContributorAutoConfiguration	javadoc
DynatraceMetricsExportAutoConfiguration	javadoc
ElasticMetricsExportAutoConfiguration	javadoc
ElasticsearchReactiveHealthContributorAutoConfiguration	javadoc
ElasticsearchRestHealthContributorAutoConfiguration	javadoc
EndpointAutoConfiguration	javadoc
EnvironmentEndpointAutoConfiguration	javadoc
FlywayEndpointAutoConfiguration	javadoc
GangliaMetricsExportAutoConfiguration	javadoc
GraphQlObservationAutoConfiguration	javadoc
GraphiteMetricsExportAutoConfiguration	javadoc
HazelcastHealthContributorAutoConfiguration	javadoc
HealthContributorAutoConfiguration	javadoc
HealthEndpointAutoConfiguration	javadoc
HeapDumpWebEndpointAutoConfiguration	javadoc
HibernateMetricsAutoConfiguration	javadoc
HttpClientObservationsAutoConfiguration	javadoc
HttpExchangesAutoConfiguration	javadoc
HttpExchangesEndpointAutoConfiguration	javadoc
HumioMetricsExportAutoConfiguration	javadoc
InfluxDbHealthContributorAutoConfiguration	javadoc
InfluxMetricsExportAutoConfiguration	javadoc
InfoContributorAutoConfiguration	javadoc
InfoEndpointAutoConfiguration	javadoc
IntegrationGraphEndpointAutoConfiguration	javadoc
JacksonEndpointAutoConfiguration	javadoc
JerseyServerMetricsAutoConfiguration	javadoc
JettyMetricsAutoConfiguration	javadoc
JmsHealthContributorAutoConfiguration	javadoc
JmxEndpointAutoConfiguration	javadoc
JmxMetricsExportAutoConfiguration	javadoc
JvmMetricsAutoConfiguration	javadoc
KafkaMetricsAutoConfiguration	javadoc

Configuration Class	Links
KairosMetricsExportAutoConfiguration	javadoc
LdapHealthContributorAutoConfiguration	javadoc
LettuceMetricsAutoConfiguration	javadoc
LiquibaseEndpointAutoConfiguration	javadoc
Log4J2MetricsAutoConfiguration	javadoc
LogFileWebEndpointAutoConfiguration	javadoc
LogbackMetricsAutoConfiguration	javadoc
LoggersEndpointAutoConfiguration	javadoc
MailHealthContributorAutoConfiguration	javadoc
ManagementContextAutoConfiguration	javadoc
ManagementWebSecurityAutoConfiguration	javadoc
MappingsEndpointAutoConfiguration	javadoc
MetricsAutoConfiguration	javadoc
MetricsEndpointAutoConfiguration	javadoc
MicrometerTracingAutoConfiguration	javadoc
MongoHealthContributorAutoConfiguration	javadoc
MongoMetricsAutoConfiguration	javadoc
MongoReactiveHealthContributorAutoConfiguration	javadoc
Neo4jHealthContributorAutoConfiguration	javadoc
NewRelicMetricsExportAutoConfiguration	javadoc
ObservationAutoConfiguration	javadoc
OpenTelemetryAutoConfiguration	javadoc
OtlpAutoConfiguration	javadoc
OtlpMetricsExportAutoConfiguration	javadoc
PrometheusExemplarsAutoConfiguration	javadoc
PrometheusMetricsExportAutoConfiguration	javadoc
QuartzEndpointAutoConfiguration	javadoc
RabbitHealthContributorAutoConfiguration	javadoc
RabbitMetricsAutoConfiguration	javadoc
ReactiveCloudFoundryActuatorAutoConfiguration	javadoc
ReactiveManagementContextAutoConfiguration	javadoc
ReactiveManagementWebSecurityAutoConfiguration	javadoc
RedisHealthContributorAutoConfiguration	javadoc

Configuration Class	Links
RedisReactiveHealthContributorAutoConfiguration	javadoc
RepositoryMetricsAutoConfiguration	javadoc
ScheduledTasksEndpointAutoConfiguration	javadoc
ServletManagementContextAutoConfiguration	javadoc
SessionsEndpointAutoConfiguration	javadoc
ShutdownEndpointAutoConfiguration	javadoc
SignalFxMetricsExportAutoConfiguration	javadoc
SimpleMetricsExportAutoConfiguration	javadoc
StackdriverMetricsExportAutoConfiguration	javadoc
StartupEndpointAutoConfiguration	javadoc
StartupTimeMetricsListenerAutoConfiguration	javadoc
StatsdMetricsExportAutoConfiguration	javadoc
SystemMetricsAutoConfiguration	javadoc
TaskExecutorMetricsAutoConfiguration	javadoc
ThreadDumpEndpointAutoConfiguration	javadoc
TomcatMetricsAutoConfiguration	javadoc
WavefrontAutoConfiguration	javadoc
WavefrontMetricsExportAutoConfiguration	javadoc
WavefrontTracingAutoConfiguration	javadoc
WebEndpointAutoConfiguration	javadoc
WebFluxObservationAutoConfiguration	javadoc
WebMvcObservationAutoConfiguration	javadoc
ZipkinAutoConfiguration	javadoc

Appendix D: Test Auto-configuration Annotations

This appendix describes the `@…Test` auto-configuration annotations that Spring Boot provides to test slices of your application.

.D.1. Test Slices

The following table lists the various `@…Test` annotations that can be used to test slices of your application and the auto-configuration that they import by default:

Test slice	Imported auto-configuration

`@DataCassandraTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.ssl.SslAutoConfiguration`

`@DataCouchbaseTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.ssl.SslAutoConfiguration`

Test slice	Imported auto-configuration

`@DataElasticsearchTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.elasticsearch.ReactiveElasticsearchRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.elasticsearch.ElasticsearchClientAutoConfiguration`
	`org.springframework.boot.autoconfigure.elasticsearch.ElasticsearchRestClientAutoConfiguration`
	`org.springframework.boot.autoconfigure.elasticsearch.ReactiveElasticsearchClientAutoConfiguration`
	`org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration`
	`org.springframework.boot.autoconfigure.ssl.SslAutoConfiguration`

Test slice	Imported auto-configuration

`@DataJdbcTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration`
	`org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration`
	`org.springframework.boot.autoconfigure.sql.init.SqlInitializationAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration`

Test slice	Imported auto-configuration

`@DataJpaTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration`
	`org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration`
	`org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration`
	`org.springframework.boot.autoconfigure.sql.init.SqlInitializationAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManagerAutoConfiguration`

`@DataLdapTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration`
	`org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration`

Test slice	Imported auto-configuration

`@DataMongoTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration`
	`org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration`
	`org.springframework.boot.autoconfigure.ssl.SslAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`

`@DataNeo4jTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.neo4j.Neo4jReactiveDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.neo4j.Neo4jReactiveRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.neo4j.Neo4jAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`

Test slice	Imported auto-configuration

`@DataR2dbcTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.r2dbc.R2dbcDataAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.r2dbc.R2dbcRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration`
	`org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration`
	`org.springframework.boot.autoconfigure.r2dbc.R2dbcAutoConfiguration`
	`org.springframework.boot.autoconfigure.r2dbc.R2dbcTransactionManagerAutoConfiguration`
	`org.springframework.boot.autoconfigure.sql.init.SqlInitializationAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`

`@DataRedisTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.redis.RedisReactiveAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration`
	`org.springframework.boot.autoconfigure.ssl.SslAutoConfiguration`

`@GraphQlTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.graphql.GraphQlAutoConfiguration`
	`org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration`
	`org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration`
	`org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration`
	`org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.graphql.tester.GraphQlTesterAutoConfiguration`

Test slice	Imported auto-configuration

`@JdbcTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration`
	`org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration`
	`org.springframework.boot.autoconfigure.sql.init.SqlInitializationAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration`

`@JooqTest`	`optional:org.springframework.boot.testcontainers.service.connection.ServiceConnectionAutoConfiguration`
	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration`
	`org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration`
	`org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration`
	`org.springframework.boot.autoconfigure.sql.init.SqlInitializationAutoConfiguration`
	`org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration`

Test slice	Imported auto-configuration

`@JsonTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.json.JsonTestersAutoConfiguration`

`@RestClientTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration`
	`org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration`
	`org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration`
	`org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.client.MockRestServiceServerAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.client.WebClientRestTemplateAutoConfiguration`

Test slice	Imported auto-configuration

`@WebFluxTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration`
	`org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration`
	`org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration`
	`org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration`
	`org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.security.oauth2.client.reactive.ReactiveOAuth2ClientAutoConfiguration`
	`org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.ReactiveOAuth2ResourceServerAutoConfiguration`
	`org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoConfiguration`
	`org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration`
	`org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration`
	`org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.reactive.WebTestClientAutoConfiguration`

Test slice	Imported auto-configuration

`@WebMvcTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration`
	`org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration`
	`org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration`
	`org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration`
	`org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration`
	`org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration`
	`org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration`
	`org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`
	`org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration`
	`org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.security.oauth2.client.servlet.OAuth2ClientAutoConfiguration`
	`org.springframework.boot.autoconfigure.security.oauth2.resource.servlet.OAuth2ResourceServerAutoConfiguration`
	`org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration`
	`org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration`
	`org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration`
	`org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration`
	`org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration`
	`org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration`
	`org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.reactive.WebTestClientAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.servlet.MockMvcAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.web.servlet.MockMvcSecurityConfiguration`
	`org.springframework.boot.test.autoconfigure.web.servlet.MockMvcWebClientAutoConfiguration`

Test slice	Imported auto-configuration

`@WebServiceClientTest`	`org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration`
	`org.springframework.boot.autoconfigure.webservices.client.WebServiceTemplateAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.webservices.client.MockWebServiceServerAutoConfiguration`
	`org.springframework.boot.test.autoconfigure.webservices.client.WebServiceClientTemplateAutoConfiguration`

`@WebServiceServerTest`	`org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration` `org.springframework.boot.test.autoconfigure.webservices.server.MockWebServiceClientAutoConfiguration`

Appendix E: The Executable Jar Format

The `spring-boot-loader` modules lets Spring Boot support executable jar and war files. If you use the Maven plugin or the Gradle plugin, executable jars are automatically generated, and you generally do not need to know the details of how they work.

If you need to create executable jars from a different build system or if you are just curious about the underlying technology, this appendix provides some background.

.E.1. Nested JARs

Java does not provide any standard way to load nested jar files (that is, jar files that are themselves contained within a jar). This can be problematic if you need to distribute a self-contained application that can be run from the command line without unpacking.

To solve this problem, many developers use “shaded” jars. A shaded jar packages all classes, from all jars, into a single “uber jar”. The problem with shaded jars is that it becomes hard to see which libraries are actually in your application. It can also be problematic if the same filename is used (but with different content) in multiple jars. Spring Boot takes a different approach and lets you actually nest jars directly.

The Executable Jar File Structure

Spring Boot Loader-compatible jar files should be structured in the following way:

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-BOOT-INF
    +-classes
    |  +-mycompany
    |     +-project
    |        +-YourClasses.class
    +-lib
       +-dependency1.jar
       +-dependency2.jar
```

Application classes should be placed in a nested `BOOT-INF/classes` directory. Dependencies should be placed in a nested `BOOT-INF/lib` directory.

The Executable War File Structure

Spring Boot Loader-compatible war files should be structured in the following way:

```
example.war
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-WEB-INF
    +-classes
    |  +-com
    |     +-mycompany
    |        +-project
    |           +-YourClasses.class
    +-lib
    |  +-dependency1.jar
    |  +-dependency2.jar
    +-lib-provided
       +-servlet-api.jar
       +-dependency3.jar
```

Dependencies should be placed in a nested `WEB-INF/lib` directory. Any dependencies that are

required when running embedded but are not required when deploying to a traditional web container should be placed in `WEB-INF/lib-provided`.

Index Files

Spring Boot Loader-compatible jar and war archives can include additional index files under the `BOOT-INF/` directory. A `classpath.idx` file can be provided for both jars and wars, and it provides the ordering that jars should be added to the classpath. The `layers.idx` file can be used only for jars, and it allows a jar to be split into logical layers for Docker/OCI image creation.

Index files follow a YAML compatible syntax so that they can be easily parsed by third-party tools. These files, however, are *not* parsed internally as YAML and they must be written in exactly the formats described below in order to be used.

Classpath Index

The classpath index file can be provided in `BOOT-INF/classpath.idx`. Typically, it is generated automatically by Spring Boot's Maven and Gradle build plugins. It provides a list of jar names (including the directory) in the order that they should be added to the classpath. When generated by the build plugins, this classpath ordering matches that used by the build system for running and testing the application. Each line must start with dash space (`"- "`) and names must be in double quotes.

For example, given the following jar:

```
example.jar
 |
 +-META-INF
 |  +-...
 +-BOOT-INF
    +-classes
    |  +...
    +-lib
       +-dependency1.jar
       +-dependency2.jar
```

The index file would look like this:

```
- "BOOT-INF/lib/dependency2.jar"
- "BOOT-INF/lib/dependency1.jar"
```

Layer Index

The layers index file can be provided in `BOOT-INF/layers.idx`. It provides a list of layers and the parts of the jar that should be contained within them. Layers are written in the order that they should be added to the Docker/OCI image. Layers names are written as quoted strings prefixed with dash space (`"- "`) and with a colon (`":"`) suffix. Layer content is either a file or directory name written as a quoted string prefixed by space space dash space (`"  - "`). A directory name ends with

/, a file name does not. When a directory name is used it means that all files inside that directory are in the same layer.

A typical example of a layers index would be:

```
- "dependencies":
  - "BOOT-INF/lib/dependency1.jar"
  - "BOOT-INF/lib/dependency2.jar"
- "application":
  - "BOOT-INF/classes/"
  - "META-INF/"
```

.E.2. Spring Boot's "JarFile" Class

The core class used to support loading nested jars is `org.springframework.boot.loader.jar.JarFile`. It lets you load jar content from a standard jar file or from nested child jar data. When first loaded, the location of each `JarEntry` is mapped to a physical file offset of the outer jar, as shown in the following example:

```
myapp.jar
+-------------------+-------------------------+
| /BOOT-INF/classes | /BOOT-INF/lib/mylib.jar |
|+-----------------+||+-----------+----------+|
||     A.class      ||||  B.class  |  C.class ||
|+-----------------+||+-----------+----------+|
+-------------------+-------------------------+
 ^                   ^           ^
 0063                3452        3980
```

The preceding example shows how `A.class` can be found in `/BOOT-INF/classes` in `myapp.jar` at position `0063`. `B.class` from the nested jar can actually be found in `myapp.jar` at position `3452`, and `C.class` is at position `3980`.

Armed with this information, we can load specific nested entries by seeking to the appropriate part of the outer jar. We do not need to unpack the archive, and we do not need to read all entry data into memory.

Compatibility With the Standard Java "JarFile"

Spring Boot Loader strives to remain compatible with existing code and libraries. `org.springframework.boot.loader.jar.JarFile` extends from `java.util.jar.JarFile` and should work as a drop-in replacement. The `getURL()` method returns a `URL` that opens a connection compatible with `java.net.JarURLConnection` and can be used with Java's `URLClassLoader`.

.E.3. Launching Executable Jars

The `org.springframework.boot.loader.Launcher` class is a special bootstrap class that is used as an executable jar's main entry point. It is the actual `Main-Class` in your jar file, and it is used to setup

an appropriate `URLClassLoader` and ultimately call your `main()` method.

There are three launcher subclasses (`JarLauncher`, `WarLauncher`, and `PropertiesLauncher`). Their purpose is to load resources (`.class` files and so on) from nested jar files or war files in directories (as opposed to those explicitly on the classpath). In the case of `JarLauncher` and `WarLauncher`, the nested paths are fixed. `JarLauncher` looks in `BOOT-INF/lib/`, and `WarLauncher` looks in `WEB-INF/lib/` and `WEB-INF/lib-provided/`. You can add extra jars in those locations if you want more. The `PropertiesLauncher` looks in `BOOT-INF/lib/` in your application archive by default. You can add additional locations by setting an environment variable called `LOADER_PATH` or `loader.path` in `loader.properties` (which is a comma-separated list of directories, archives, or directories within archives).

Launcher Manifest

You need to specify an appropriate `Launcher` as the `Main-Class` attribute of `META-INF/MANIFEST.MF`. The actual class that you want to launch (that is, the class that contains a `main` method) should be specified in the `Start-Class` attribute.

The following example shows a typical `MANIFEST.MF` for an executable jar file:

```
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

For a war file, it would be as follows:

```
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```

NOTE	You need not specify `Class-Path` entries in your manifest file. The classpath is deduced from the nested jars.

.E.4. PropertiesLauncher Features

`PropertiesLauncher` has a few special features that can be enabled with external properties (System properties, environment variables, manifest entries, or `loader.properties`). The following table describes these properties:

Key	Purpose
`loader.path`	Comma-separated Classpath, such as `lib,${HOME}/app/lib`. Earlier entries take precedence, like a regular `-classpath` on the `javac` command line.

Key	Purpose
`loader.home`	Used to resolve relative paths in `loader.path`. For example, given `loader.path=lib`, then `${loader.home}/lib` is a classpath location (along with all jar files in that directory). This property is also used to locate a `loader.properties` file, as in the following example `/opt/app` It defaults to `${user.dir}`.
`loader.args`	Default arguments for the main method (space separated).
`loader.main`	Name of main class to launch (for example, `com.app.Application`).
`loader.config.name`	Name of properties file (for example, `launcher`). It defaults to `loader`.
`loader.config.location`	Path to properties file (for example, `classpath:loader.properties`). It defaults to `loader.properties`.
`loader.system`	Boolean flag to indicate that all properties should be added to System properties. It defaults to `false`.

When specified as environment variables or manifest entries, the following names should be used:

Key	Manifest entry	Environment variable
`loader.path`	`Loader-Path`	`LOADER_PATH`
`loader.home`	`Loader-Home`	`LOADER_HOME`
`loader.args`	`Loader-Args`	`LOADER_ARGS`
`loader.main`	`Start-Class`	`LOADER_MAIN`
`loader.config.location`	`Loader-Config-Location`	`LOADER_CONFIG_LOCATION`
`loader.system`	`Loader-System`	`LOADER_SYSTEM`

TIP Build plugins automatically move the `Main-Class` attribute to `Start-Class` when the fat jar is built. If you use that, specify the name of the class to launch by using the `Main-Class` attribute and leaving out `Start-Class`.

The following rules apply to working with `PropertiesLauncher`:

- `loader.properties` is searched for in `loader.home`, then in the root of the classpath, and then in `classpath:/BOOT-INF/classes`. The first location where a file with that name exists is used.
- `loader.home` is the directory location of an additional properties file (overriding the default) only when `loader.config.location` is not specified.
- `loader.path` can contain directories (which are scanned recursively for jar and zip files), archive paths, a directory within an archive that is scanned for jar files (for example,

`dependencies.jar!/lib`), or wildcard patterns (for the default JVM behavior). Archive paths can be relative to `loader.home` or anywhere in the file system with a `jar:file:` prefix.

- `loader.path` (if empty) defaults to `BOOT-INF/lib` (meaning a local directory or a nested one if running from an archive). Because of this, `PropertiesLauncher` behaves the same as `JarLauncher` when no additional configuration is provided.
- `loader.path` can not be used to configure the location of `loader.properties` (the classpath used to search for the latter is the JVM classpath when `PropertiesLauncher` is launched).
- Placeholder replacement is done from System and environment variables plus the properties file itself on all values before use.
- The search order for properties (where it makes sense to look in more than one place) is environment variables, system properties, `loader.properties`, the exploded archive manifest, and the archive manifest.

.E.5. Executable Jar Restrictions

You need to consider the following restrictions when working with a Spring Boot Loader packaged application:

- Zip entry compression: The `ZipEntry` for a nested jar must be saved by using the `ZipEntry.STORED` method. This is required so that we can seek directly to individual content within the nested jar. The content of the nested jar file itself can still be compressed, as can any other entries in the outer jar.
- System classLoader: Launched applications should use `Thread.getContextClassLoader()` when loading classes (most libraries and frameworks do so by default). Trying to load nested jar classes with `ClassLoader.getSystemClassLoader()` fails. `java.util.Logging` always uses the system classloader. For this reason, you should consider a different logging implementation.

.E.6. Alternative Single Jar Solutions

If the preceding restrictions mean that you cannot use Spring Boot Loader, consider the following alternatives:

- Maven Shade Plugin
- JarClassLoader
- OneJar
- Gradle Shadow Plugin

Appendix F: Dependency Versions

This appendix provides details of the dependencies that are managed by Spring Boot.

.F.1. Managed Dependency Coordinates

The following table provides details of all of the dependency versions that are provided by Spring Boot in its CLI (Command Line Interface), Maven dependency management, and Gradle plugin.

When you declare a dependency on one of these artifacts without declaring a version, the version listed in the table is used.

Group ID	Artifact ID	Version
ch.qos.logback	logback-access	1.4.11
ch.qos.logback	logback-classic	1.4.11
ch.qos.logback	logback-core	1.4.11
co.elastic.clients	elasticsearch-java	8.7.1
com.couchbase.client	java-client	3.4.11
com.datastax.oss	java-driver-core	4.15.0
com.datastax.oss	java-driver-core-shaded	4.15.0
com.datastax.oss	java-driver-mapper-processor	4.15.0
com.datastax.oss	java-driver-mapper-runtime	4.15.0
com.datastax.oss	java-driver-metrics-micrometer	4.15.0
com.datastax.oss	java-driver-metrics-microprofile	4.15.0
com.datastax.oss	java-driver-query-builder	4.15.0
com.datastax.oss	java-driver-shaded-guava	25.1-jre-graal-sub-1
com.datastax.oss	java-driver-test-infra	4.15.0
com.datastax.oss	native-protocol	1.5.1
com.fasterxml	classmate	1.5.1
com.fasterxml.jackson.core	jackson-annotations	2.15.3
com.fasterxml.jackson.core	jackson-core	2.15.3
com.fasterxml.jackson.core	jackson-databind	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-avro	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-cbor	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-csv	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-ion	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-properties	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-protobuf	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-smile	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-toml	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-xml	2.15.3
com.fasterxml.jackson.dataformat	jackson-dataformat-yaml	2.15.3

Group ID	Artifact ID	Version
com.fasterxml.jackson.datatype	jackson-datatype-eclipse-collections	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-guava	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-hibernate4	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-hibernate5	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-hibernate5-jakarta	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-hibernate6	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-hppc	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-jakarta-jsonp	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-jaxrs	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-jdk8	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-joda	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-joda-money	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-json-org	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-jsr310	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-jsr353	2.15.3
com.fasterxml.jackson.datatype	jackson-datatype-pcollections	2.15.3
com.fasterxml.jackson.jakarta.rs	jackson-jakarta-rs-base	2.15.3
com.fasterxml.jackson.jakarta.rs	jackson-jakarta-rs-cbor-provider	2.15.3
com.fasterxml.jackson.jakarta.rs	jackson-jakarta-rs-json-provider	2.15.3
com.fasterxml.jackson.jakarta.rs	jackson-jakarta-rs-smile-provider	2.15.3
com.fasterxml.jackson.jakarta.rs	jackson-jakarta-rs-xml-provider	2.15.3
com.fasterxml.jackson.jakarta.rs	jackson-jakarta-rs-yaml-provider	2.15.3
com.fasterxml.jackson.jaxrs	jackson-jaxrs-base	2.15.3
com.fasterxml.jackson.jaxrs	jackson-jaxrs-cbor-provider	2.15.3
com.fasterxml.jackson.jaxrs	jackson-jaxrs-json-provider	2.15.3
com.fasterxml.jackson.jaxrs	jackson-jaxrs-smile-provider	2.15.3
com.fasterxml.jackson.jaxrs	jackson-jaxrs-xml-provider	2.15.3
com.fasterxml.jackson.jaxrs	jackson-jaxrs-yaml-provider	2.15.3
com.fasterxml.jackson.jr	jackson-jr-all	2.15.3
com.fasterxml.jackson.jr	jackson-jr-annotation-support	2.15.3
com.fasterxml.jackson.jr	jackson-jr-objects	2.15.3
com.fasterxml.jackson.jr	jackson-jr-retrofit2	2.15.3
com.fasterxml.jackson.jr	jackson-jr-stree	2.15.3

Group ID	Artifact ID	Version
com.fasterxml.jackson.module	jackson-module-afterburner	2.15.3
com.fasterxml.jackson.module	jackson-module-blackbird	2.15.3
com.fasterxml.jackson.module	jackson-module-guice	2.15.3
com.fasterxml.jackson.module	jackson-module-jakarta-xmlbind-annotations	2.15.3
com.fasterxml.jackson.module	jackson-module-jaxb-annotations	2.15.3
com.fasterxml.jackson.module	jackson-module-jsonSchema	2.15.3
com.fasterxml.jackson.module	jackson-module-jsonSchema-jakarta	2.15.3
com.fasterxml.jackson.module	jackson-module-kotlin	2.15.3
com.fasterxml.jackson.module	jackson-module-mrbean	2.15.3
com.fasterxml.jackson.module	jackson-module-no-ctor-deser	2.15.3
com.fasterxml.jackson.module	jackson-module-osgi	2.15.3
com.fasterxml.jackson.module	jackson-module-parameter-names	2.15.3
com.fasterxml.jackson.module	jackson-module-paranamer	2.15.3
com.fasterxml.jackson.module	jackson-module-scala_2.11	2.15.3
com.fasterxml.jackson.module	jackson-module-scala_2.12	2.15.3
com.fasterxml.jackson.module	jackson-module-scala_2.13	2.15.3
com.fasterxml.jackson.module	jackson-module-scala_3	2.15.3
com.github.ben-manes.caffeine	caffeine	3.1.8
com.github.ben-manes.caffeine	guava	3.1.8
com.github.ben-manes.caffeine	jcache	3.1.8
com.github.ben-manes.caffeine	simulator	3.1.8
com.github.mxab.thymeleaf.extras	thymeleaf-extras-data-attribute	2.0.1
com.google.code.gson	gson	2.10.1
com.graphql-java	graphql-java	20.2
com.h2database	h2	2.1.214
com.hazelcast	hazelcast	5.2.4
com.hazelcast	hazelcast-spring	5.2.4
com.ibm.db2	jcc	11.5.9.0
com.jayway.jsonpath	json-path	2.8.0
com.jayway.jsonpath	json-path-assert	2.8.0
com.microsoft.sqlserver	mssql-jdbc	11.2.3.jre17
com.mysql	mysql-connector-j	8.0.33
com.oracle.database.ha	ons	21.9.0.0
com.oracle.database.ha	simplefan	21.9.0.0
com.oracle.database.jdbc	ojdbc11	21.9.0.0
com.oracle.database.jdbc	ojdbc11-production	21.9.0.0

Group ID	Artifact ID	Version
com.oracle.database.jdbc	ojdbc8	21.9.0.0
com.oracle.database.jdbc	ojdbc8-production	21.9.0.0
com.oracle.database.jdbc	rsi	21.9.0.0
com.oracle.database.jdbc	ucp	21.9.0.0
com.oracle.database.jdbc	ucp11	21.9.0.0
com.oracle.database.jdbc.debug	ojdbc11-debug	21.9.0.0
com.oracle.database.jdbc.debug	ojdbc11-observability-debug	21.9.0.0
com.oracle.database.jdbc.debug	ojdbc11_g	21.9.0.0
com.oracle.database.jdbc.debug	ojdbc11dms_g	21.9.0.0
com.oracle.database.jdbc.debug	ojdbc8-debug	21.9.0.0
com.oracle.database.jdbc.debug	ojdbc8-observability-debug	21.9.0.0
com.oracle.database.jdbc.debug	ojdbc8_g	21.9.0.0
com.oracle.database.jdbc.debug	ojdbc8dms_g	21.9.0.0
com.oracle.database.nls	orai18n	21.9.0.0
com.oracle.database.observability	dms	21.9.0.0
com.oracle.database.observability	ojdbc11-observability	21.9.0.0
com.oracle.database.observability	ojdbc11dms	21.9.0.0
com.oracle.database.observability	ojdbc8-observability	21.9.0.0
com.oracle.database.observability	ojdbc8dms	21.9.0.0
com.oracle.database.r2dbc	oracle-r2dbc	1.1.1
com.oracle.database.security	oraclepki	21.9.0.0
com.oracle.database.security	osdt_cert	21.9.0.0
com.oracle.database.security	osdt_core	21.9.0.0
com.oracle.database.xml	xdb	21.9.0.0
com.oracle.database.xml	xmlparserv2	21.9.0.0
com.querydsl	querydsl-apt	5.0.0
com.querydsl	querydsl-codegen	5.0.0
com.querydsl	querydsl-codegen-utils	5.0.0
com.querydsl	querydsl-collections	5.0.0
com.querydsl	querydsl-core	5.0.0
com.querydsl	querydsl-guava	5.0.0
com.querydsl	querydsl-hibernate-search	5.0.0
com.querydsl	querydsl-jdo	5.0.0
com.querydsl	querydsl-jpa	5.0.0
com.querydsl	querydsl-jpa-codegen	5.0.0

Group ID	Artifact ID	Version
com.querydsl	querydsl-kotlin	5.0.0
com.querydsl	querydsl-kotlin-codegen	5.0.0
com.querydsl	querydsl-lucene3	5.0.0
com.querydsl	querydsl-lucene4	5.0.0
com.querydsl	querydsl-lucene5	5.0.0
com.querydsl	querydsl-mongodb	5.0.0
com.querydsl	querydsl-scala	5.0.0
com.querydsl	querydsl-spatial	5.0.0
com.querydsl	querydsl-sql	5.0.0
com.querydsl	querydsl-sql-codegen	5.0.0
com.querydsl	querydsl-sql-spatial	5.0.0
com.querydsl	querydsl-sql-spring	5.0.0
com.rabbitmq	amqp-client	5.17.1
com.rabbitmq	stream-client	0.9.0
com.samskivert	jmustache	1.15
com.sendgrid	sendgrid-java	4.9.3
com.squareup.okhttp3	logging-interceptor	4.10.0
com.squareup.okhttp3	mockwebserver	4.10.0
com.squareup.okhttp3	okcurl	4.10.0
com.squareup.okhttp3	okhttp	4.10.0
com.squareup.okhttp3	okhttp-brotli	4.10.0
com.squareup.okhttp3	okhttp-dnsoverhttps	4.10.0
com.squareup.okhttp3	okhttp-sse	4.10.0
com.squareup.okhttp3	okhttp-tls	4.10.0
com.squareup.okhttp3	okhttp-urlconnection	4.10.0
com.sun.istack	istack-commons-runtime	4.1.2
com.sun.xml.bind	jaxb-core	4.0.4
com.sun.xml.bind	jaxb-impl	4.0.4
com.sun.xml.bind	jaxb-jxc	4.0.4
com.sun.xml.bind	jaxb-osgi	4.0.4
com.sun.xml.bind	jaxb-xjc	4.0.4
com.sun.xml.fastinfoset	FastInfoset	2.1.0
com.sun.xml.messaging.saaj	saaj-impl	3.0.3
com.unboundid	unboundid-ldapsdk	6.0.10
com.zaxxer	HikariCP	5.0.1
commons-codec	commons-codec	1.15
commons-pool	commons-pool	1.6
io.asyncer	r2dbc-mysql	1.0.5

Group ID	Artifact ID	Version
io.dropwizard.metrics	metrics-annotation	4.2.22
io.dropwizard.metrics	metrics-caffeine	4.2.22
io.dropwizard.metrics	metrics-caffeine3	4.2.22
io.dropwizard.metrics	metrics-collectd	4.2.22
io.dropwizard.metrics	metrics-core	4.2.22
io.dropwizard.metrics	metrics-ehcache	4.2.22
io.dropwizard.metrics	metrics-graphite	4.2.22
io.dropwizard.metrics	metrics-healthchecks	4.2.22
io.dropwizard.metrics	metrics-httpasyncclient	4.2.22
io.dropwizard.metrics	metrics-httpclient	4.2.22
io.dropwizard.metrics	metrics-httpclient5	4.2.22
io.dropwizard.metrics	metrics-jakarta-servlet	4.2.22
io.dropwizard.metrics	metrics-jakarta-servlet6	4.2.22
io.dropwizard.metrics	metrics-jakarta-servlets	4.2.22
io.dropwizard.metrics	metrics-jcache	4.2.22
io.dropwizard.metrics	metrics-jdbi	4.2.22
io.dropwizard.metrics	metrics-jdbi3	4.2.22
io.dropwizard.metrics	metrics-jersey2	4.2.22
io.dropwizard.metrics	metrics-jersey3	4.2.22
io.dropwizard.metrics	metrics-jersey31	4.2.22
io.dropwizard.metrics	metrics-jetty10	4.2.22
io.dropwizard.metrics	metrics-jetty11	4.2.22
io.dropwizard.metrics	metrics-jetty12	4.2.22
io.dropwizard.metrics	metrics-jetty12-ee10	4.2.22
io.dropwizard.metrics	metrics-jetty9	4.2.22
io.dropwizard.metrics	metrics-jmx	4.2.22
io.dropwizard.metrics	metrics-json	4.2.22
io.dropwizard.metrics	metrics-jvm	4.2.22
io.dropwizard.metrics	metrics-log4j2	4.2.22
io.dropwizard.metrics	metrics-logback	4.2.22
io.dropwizard.metrics	metrics-logback13	4.2.22
io.dropwizard.metrics	metrics-logback14	4.2.22
io.dropwizard.metrics	metrics-servlet	4.2.22
io.dropwizard.metrics	metrics-servlets	4.2.22
io.lettuce	lettuce-core	6.2.7.RELEASE
io.micrometer	benchmarks	1.1.7
io.micrometer	micrometer-commons	1.11.6
io.micrometer	micrometer-core	1.11.6

Group ID	Artifact ID	Version
io.micrometer	micrometer-jetty11	1.11.6
io.micrometer	micrometer-observation	1.11.6
io.micrometer	micrometer-observation-test	1.11.6
io.micrometer	micrometer-registry-appoptics	1.11.6
io.micrometer	micrometer-registry-atlas	1.11.6
io.micrometer	micrometer-registry-azure-monitor	1.11.6
io.micrometer	micrometer-registry-cloudwatch	1.11.6
io.micrometer	micrometer-registry-cloudwatch2	1.11.6
io.micrometer	micrometer-registry-datadog	1.11.6
io.micrometer	micrometer-registry-dynatrace	1.11.6
io.micrometer	micrometer-registry-elastic	1.11.6
io.micrometer	micrometer-registry-ganglia	1.11.6
io.micrometer	micrometer-registry-graphite	1.11.6
io.micrometer	micrometer-registry-health	1.11.6
io.micrometer	micrometer-registry-humio	1.11.6
io.micrometer	micrometer-registry-influx	1.11.6
io.micrometer	micrometer-registry-jmx	1.11.6
io.micrometer	micrometer-registry-kairos	1.11.6
io.micrometer	micrometer-registry-new-relic	1.11.6
io.micrometer	micrometer-registry-opentsdb	1.11.6
io.micrometer	micrometer-registry-otlp	1.11.6
io.micrometer	micrometer-registry-prometheus	1.11.6
io.micrometer	micrometer-registry-signalfx	1.11.6
io.micrometer	micrometer-registry-stackdriver	1.11.6
io.micrometer	micrometer-registry-statsd	1.11.6
io.micrometer	micrometer-registry-wavefront	1.11.6
io.micrometer	micrometer-test	1.11.6
io.micrometer	micrometer-tracing	1.1.7
io.micrometer	micrometer-tracing-bridge-brave	1.1.7
io.micrometer	micrometer-tracing-bridge-otel	1.1.7
io.micrometer	micrometer-tracing-integration-test	1.1.7
io.micrometer	micrometer-tracing-reporter-wavefront	1.1.7
io.micrometer	micrometer-tracing-test	1.1.7
io.netty	netty-all	4.1.101.Final

Group ID	Artifact ID	Version
io.netty	netty-buffer	4.1.101.Final
io.netty	netty-codec	4.1.101.Final
io.netty	netty-codec-dns	4.1.101.Final
io.netty	netty-codec-haproxy	4.1.101.Final
io.netty	netty-codec-http	4.1.101.Final
io.netty	netty-codec-http2	4.1.101.Final
io.netty	netty-codec-memcache	4.1.101.Final
io.netty	netty-codec-mqtt	4.1.101.Final
io.netty	netty-codec-redis	4.1.101.Final
io.netty	netty-codec-smtp	4.1.101.Final
io.netty	netty-codec-socks	4.1.101.Final
io.netty	netty-codec-stomp	4.1.101.Final
io.netty	netty-codec-xml	4.1.101.Final
io.netty	netty-common	4.1.101.Final
io.netty	netty-dev-tools	4.1.101.Final
io.netty	netty-example	4.1.101.Final
io.netty	netty-handler	4.1.101.Final
io.netty	netty-handler-proxy	4.1.101.Final
io.netty	netty-handler-ssl-ocsp	4.1.101.Final
io.netty	netty-resolver	4.1.101.Final
io.netty	netty-resolver-dns	4.1.101.Final
io.netty	netty-resolver-dns-classes-macos	4.1.101.Final
io.netty	netty-resolver-dns-native-macos	4.1.101.Final
io.netty	netty-tcnative	2.0.61.Final
io.netty	netty-tcnative-boringssl-static	2.0.61.Final
io.netty	netty-tcnative-classes	2.0.61.Final
io.netty	netty-transport	4.1.101.Final
io.netty	netty-transport-classes-epoll	4.1.101.Final
io.netty	netty-transport-classes-kqueue	4.1.101.Final
io.netty	netty-transport-native-epoll	4.1.101.Final
io.netty	netty-transport-native-kqueue	4.1.101.Final
io.netty	netty-transport-native-unix-common	4.1.101.Final
io.netty	netty-transport-rxtx	4.1.101.Final
io.netty	netty-transport-sctp	4.1.101.Final
io.netty	netty-transport-udt	4.1.101.Final
io.opentelemetry	opentelemetry-api	1.25.0

Group ID	Artifact ID	Version
io.opentelemetry	opentelemetry-context	1.25.0
io.opentelemetry	opentelemetry-exporter-common	1.25.0
io.opentelemetry	opentelemetry-exporter-jaeger	1.25.0
io.opentelemetry	opentelemetry-exporter-jaeger-proto	1.17.0
io.opentelemetry	opentelemetry-exporter-jaeger-thrift	1.25.0
io.opentelemetry	opentelemetry-exporter-logging	1.25.0
io.opentelemetry	opentelemetry-exporter-logging-otlp	1.25.0
io.opentelemetry	opentelemetry-exporter-otlp	1.25.0
io.opentelemetry	opentelemetry-exporter-otlp-common	1.25.0
io.opentelemetry	opentelemetry-exporter-zipkin	1.25.0
io.opentelemetry	opentelemetry-extension-annotations	1.18.0
io.opentelemetry	opentelemetry-extension-aws	1.20.1
io.opentelemetry	opentelemetry-extension-kotlin	1.25.0
io.opentelemetry	opentelemetry-extension-trace-propagators	1.25.0
io.opentelemetry	opentelemetry-sdk	1.25.0
io.opentelemetry	opentelemetry-sdk-common	1.25.0
io.opentelemetry	opentelemetry-sdk-extension-autoconfigure-spi	1.25.0
io.opentelemetry	opentelemetry-sdk-extension-aws	1.19.0
io.opentelemetry	opentelemetry-sdk-extension-jaeger-remote-sampler	1.25.0
io.opentelemetry	opentelemetry-sdk-extension-resources	1.19.0
io.opentelemetry	opentelemetry-sdk-metrics	1.25.0
io.opentelemetry	opentelemetry-sdk-testing	1.25.0
io.opentelemetry	opentelemetry-sdk-trace	1.25.0
io.projectreactor	reactor-core	3.5.12
io.projectreactor	reactor-core-micrometer	1.0.12
io.projectreactor	reactor-test	3.5.12
io.projectreactor	reactor-tools	3.5.12
io.projectreactor.addons	reactor-adapter	3.5.1
io.projectreactor.addons	reactor-extra	3.5.1
io.projectreactor.addons	reactor-pool	1.0.3
io.projectreactor.addons	reactor-pool-micrometer	0.1.3
io.projectreactor.kafka	reactor-kafka	1.3.22

Group ID	Artifact ID	Version
io.projectreactor.kotlin	reactor-kotlin-extensions	1.2.2
io.projectreactor.netty	reactor-netty	1.1.13
io.projectreactor.netty	reactor-netty-core	1.1.13
io.projectreactor.netty	reactor-netty-http	1.1.13
io.projectreactor.netty	reactor-netty-http-brave	1.1.13
io.prometheus	simpleclient	0.16.0
io.prometheus	simpleclient_caffeine	0.16.0
io.prometheus	simpleclient_common	0.16.0
io.prometheus	simpleclient_dropwizard	0.16.0
io.prometheus	simpleclient_graphite_bridge	0.16.0
io.prometheus	simpleclient_guava	0.16.0
io.prometheus	simpleclient_hibernate	0.16.0
io.prometheus	simpleclient_hotspot	0.16.0
io.prometheus	simpleclient_httpserver	0.16.0
io.prometheus	simpleclient_jetty	0.16.0
io.prometheus	simpleclient_jetty_jdk8	0.16.0
io.prometheus	simpleclient_log4j	0.16.0
io.prometheus	simpleclient_log4j2	0.16.0
io.prometheus	simpleclient_logback	0.16.0
io.prometheus	simpleclient_pushgateway	0.16.0
io.prometheus	simpleclient_servlet	0.16.0
io.prometheus	simpleclient_servlet_jakarta	0.16.0
io.prometheus	simpleclient_spring_boot	0.16.0
io.prometheus	simpleclient_spring_web	0.16.0
io.prometheus	simpleclient_tracer_common	0.16.0
io.prometheus	simpleclient_tracer_otel	0.16.0
io.prometheus	simpleclient_tracer_otel_agent	0.16.0
io.prometheus	simpleclient_vertx	0.16.0
io.r2dbc	r2dbc-h2	1.0.0.RELEASE
io.r2dbc	r2dbc-mssql	1.0.2.RELEASE
io.r2dbc	r2dbc-pool	1.0.1.RELEASE
io.r2dbc	r2dbc-proxy	1.1.2.RELEASE
io.r2dbc	r2dbc-spi	1.0.0.RELEASE
io.reactivex.rxjava3	rxjava	3.1.8
io.rest-assured	json-path	5.3.2
io.rest-assured	json-schema-validator	5.3.2
io.rest-assured	kotlin-extensions	5.3.2
io.rest-assured	rest-assured	5.3.2

Group ID	Artifact ID	Version
io.rest-assured	rest-assured-all	5.3.2
io.rest-assured	rest-assured-common	5.3.2
io.rest-assured	scala-support	5.3.2
io.rest-assured	spring-commons	5.3.2
io.rest-assured	spring-mock-mvc	5.3.2
io.rest-assured	spring-mock-mvc-kotlin-extensions	5.3.2
io.rest-assured	spring-web-test-client	5.3.2
io.rest-assured	xml-path	5.3.2
io.rsocket	rsocket-core	1.1.3
io.rsocket	rsocket-load-balancer	1.1.3
io.rsocket	rsocket-micrometer	1.1.3
io.rsocket	rsocket-test	1.1.3
io.rsocket	rsocket-transport-local	1.1.3
io.rsocket	rsocket-transport-netty	1.1.3
io.spring.gradle	dependency-management-plugin	1.1.4
io.undertow	undertow-core	2.3.10.Final
io.undertow	undertow-servlet	2.3.10.Final
io.undertow	undertow-websockets-jsr	2.3.10.Final
io.zipkin.brave	brave	5.15.1
io.zipkin.brave	brave-context-jfr	5.15.1
io.zipkin.brave	brave-context-log4j12	5.15.1
io.zipkin.brave	brave-context-log4j2	5.15.1
io.zipkin.brave	brave-context-rxjava2	5.15.1
io.zipkin.brave	brave-context-slf4j	5.15.1
io.zipkin.brave	brave-instrumentation-dubbo	5.15.1
io.zipkin.brave	brave-instrumentation-dubbo-rpc	5.15.1
io.zipkin.brave	brave-instrumentation-grpc	5.15.1
io.zipkin.brave	brave-instrumentation-http	5.15.1
io.zipkin.brave	brave-instrumentation-http-tests	5.15.1
io.zipkin.brave	brave-instrumentation-httpasyncclient	5.15.1
io.zipkin.brave	brave-instrumentation-httpclient	5.15.1
io.zipkin.brave	brave-instrumentation-jaxrs2	5.15.1
io.zipkin.brave	brave-instrumentation-jersey-server	5.15.1
io.zipkin.brave	brave-instrumentation-jms	5.15.1

Group ID	Artifact ID	Version
io.zipkin.brave	brave-instrumentation-jms-jakarta	5.15.1
io.zipkin.brave	brave-instrumentation-kafka-clients	5.15.1
io.zipkin.brave	brave-instrumentation-kafka-streams	5.15.1
io.zipkin.brave	brave-instrumentation-messaging	5.15.1
io.zipkin.brave	brave-instrumentation-mongodb	5.15.1
io.zipkin.brave	brave-instrumentation-mysql	5.15.1
io.zipkin.brave	brave-instrumentation-mysql6	5.15.1
io.zipkin.brave	brave-instrumentation-mysql8	5.15.1
io.zipkin.brave	brave-instrumentation-netty-codec-http	5.15.1
io.zipkin.brave	brave-instrumentation-okhttp3	5.15.1
io.zipkin.brave	brave-instrumentation-p6spy	5.15.1
io.zipkin.brave	brave-instrumentation-rpc	5.15.1
io.zipkin.brave	brave-instrumentation-servlet	5.15.1
io.zipkin.brave	brave-instrumentation-servlet-jakarta	5.15.1
io.zipkin.brave	brave-instrumentation-sparkjava	5.15.1
io.zipkin.brave	brave-instrumentation-spring-rabbit	5.15.1
io.zipkin.brave	brave-instrumentation-spring-web	5.15.1
io.zipkin.brave	brave-instrumentation-spring-webmvc	5.15.1
io.zipkin.brave	brave-instrumentation-vertx-web	5.15.1
io.zipkin.brave	brave-spring-beans	5.15.1
io.zipkin.brave	brave-tests	5.15.1
io.zipkin.proto3	zipkin-proto3	1.0.0
io.zipkin.reporter2	zipkin-reporter	2.16.3
io.zipkin.reporter2	zipkin-reporter-brave	2.16.3
io.zipkin.reporter2	zipkin-reporter-metrics-micrometer	2.16.3
io.zipkin.reporter2	zipkin-reporter-spring-beans	2.16.3
io.zipkin.reporter2	zipkin-sender-activemq-client	2.16.3
io.zipkin.reporter2	zipkin-sender-amqp-client	2.16.3
io.zipkin.reporter2	zipkin-sender-kafka	2.16.3
io.zipkin.reporter2	zipkin-sender-kafka08	2.16.3

Group ID	Artifact ID	Version
io.zipkin.reporter2	zipkin-sender-libthrift	2.16.3
io.zipkin.reporter2	zipkin-sender-okhttp3	2.16.3
io.zipkin.reporter2	zipkin-sender-urlconnection	2.16.3
io.zipkin.zipkin2	zipkin	2.23.2
jakarta.activation	jakarta.activation-api	2.1.2
jakarta.annotation	jakarta.annotation-api	2.1.1
jakarta.jms	jakarta.jms-api	3.1.0
jakarta.json	jakarta.json-api	2.1.3
jakarta.json.bind	jakarta.json.bind-api	3.0.0
jakarta.mail	jakarta.mail-api	2.1.2
jakarta.management.j2ee	jakarta.management.j2ee-api	1.1.4
jakarta.persistence	jakarta.persistence-api	3.1.0
jakarta.servlet	jakarta.servlet-api	6.0.0
jakarta.servlet.jsp.jstl	jakarta.servlet.jsp.jstl-api	3.0.0
jakarta.transaction	jakarta.transaction-api	2.0.1
jakarta.validation	jakarta.validation-api	3.0.2
jakarta.websocket	jakarta.websocket-api	2.1.1
jakarta.websocket	jakarta.websocket-client-api	2.1.1
jakarta.ws.rs	jakarta.ws.rs-api	3.1.0
jakarta.xml.bind	jakarta.xml.bind-api	4.0.1
jakarta.xml.soap	jakarta.xml.soap-api	3.0.1
jakarta.xml.ws	jakarta.xml.ws-api	4.0.1
javax.cache	cache-api	1.1.1
javax.money	money-api	1.1
jaxen	jaxen	2.0.0
junit	junit	4.13.2
net.bytebuddy	byte-buddy	1.14.10
net.bytebuddy	byte-buddy-agent	1.14.10
net.minidev	json-smart	2.4.11
net.sourceforge.htmlunit	htmlunit	2.70.0
net.sourceforge.jtds	jtds	1.3.1
net.sourceforge.nekohtml	nekohtml	1.9.22
nz.net.ultraq.thymeleaf	thymeleaf-layout-dialect	3.2.1
org.apache.activemq	activemq-amqp	5.18.3
org.apache.activemq	activemq-blueprint	5.18.3
org.apache.activemq	activemq-broker	5.18.3
org.apache.activemq	activemq-client	5.18.3
org.apache.activemq	activemq-client-jakarta	5.18.3

Group ID	Artifact ID	Version
org.apache.activemq	activemq-console	5.18.3
org.apache.activemq	activemq-http	5.18.3
org.apache.activemq	activemq-jaas	5.18.3
org.apache.activemq	activemq-jdbc-store	5.18.3
org.apache.activemq	activemq-jms-pool	5.18.3
org.apache.activemq	activemq-kahadb-store	5.18.3
org.apache.activemq	activemq-karaf	5.18.3
org.apache.activemq	activemq-log4j-appender	5.18.3
org.apache.activemq	activemq-mqtt	5.18.3
org.apache.activemq	activemq-openwire-generator	5.18.3
org.apache.activemq	activemq-openwire-legacy	5.18.3
org.apache.activemq	activemq-osgi	5.18.3
org.apache.activemq	activemq-partition	5.18.3
org.apache.activemq	activemq-pool	5.18.3
org.apache.activemq	activemq-ra	5.18.3
org.apache.activemq	activemq-run	5.18.3
org.apache.activemq	activemq-runtime-config	5.18.3
org.apache.activemq	activemq-shiro	5.18.3
org.apache.activemq	activemq-spring	5.18.3
org.apache.activemq	activemq-stomp	5.18.3
org.apache.activemq	activemq-web	5.18.3
org.apache.activemq	artemis-amqp-protocol	2.28.0
org.apache.activemq	artemis-commons	2.28.0
org.apache.activemq	artemis-core-client	2.28.0
org.apache.activemq	artemis-jakarta-client	2.28.0
org.apache.activemq	artemis-jakarta-server	2.28.0
org.apache.activemq	artemis-jakarta-service-extensions	2.28.0
org.apache.activemq	artemis-jdbc-store	2.28.0
org.apache.activemq	artemis-journal	2.28.0
org.apache.activemq	artemis-quorum-api	2.28.0
org.apache.activemq	artemis-selector	2.28.0
org.apache.activemq	artemis-server	2.28.0
org.apache.activemq	artemis-service-extensions	2.28.0
org.apache.commons	commons-dbcp2	2.9.0
org.apache.commons	commons-lang3	3.12.0
org.apache.commons	commons-pool2	2.11.1
org.apache.derby	derby	10.16.1.1
org.apache.derby	derbyclient	10.16.1.1

Group ID	Artifact ID	Version
org.apache.derby	derbynet	10.16.1.1
org.apache.derby	derbyoptionaltools	10.16.1.1
org.apache.derby	derbyshared	10.16.1.1
org.apache.derby	derbytools	10.16.1.1
org.apache.groovy	groovy	4.0.15
org.apache.groovy	groovy-ant	4.0.15
org.apache.groovy	groovy-astbuilder	4.0.15
org.apache.groovy	groovy-cli-commons	4.0.15
org.apache.groovy	groovy-cli-picocli	4.0.15
org.apache.groovy	groovy-console	4.0.15
org.apache.groovy	groovy-contracts	4.0.15
org.apache.groovy	groovy-datetime	4.0.15
org.apache.groovy	groovy-dateutil	4.0.15
org.apache.groovy	groovy-docgenerator	4.0.15
org.apache.groovy	groovy-ginq	4.0.15
org.apache.groovy	groovy-groovydoc	4.0.15
org.apache.groovy	groovy-groovysh	4.0.15
org.apache.groovy	groovy-jmx	4.0.15
org.apache.groovy	groovy-json	4.0.15
org.apache.groovy	groovy-jsr223	4.0.15
org.apache.groovy	groovy-macro	4.0.15
org.apache.groovy	groovy-macro-library	4.0.15
org.apache.groovy	groovy-nio	4.0.15
org.apache.groovy	groovy-servlet	4.0.15
org.apache.groovy	groovy-sql	4.0.15
org.apache.groovy	groovy-swing	4.0.15
org.apache.groovy	groovy-templates	4.0.15
org.apache.groovy	groovy-test	4.0.15
org.apache.groovy	groovy-test-junit5	4.0.15
org.apache.groovy	groovy-testng	4.0.15
org.apache.groovy	groovy-toml	4.0.15
org.apache.groovy	groovy-typecheckers	4.0.15
org.apache.groovy	groovy-xml	4.0.15
org.apache.groovy	groovy-yaml	4.0.15
org.apache.httpcomponents	httpasyncclient	4.1.5
org.apache.httpcomponents	httpcore	4.4.16
org.apache.httpcomponents	httpcore-nio	4.4.16
org.apache.httpcomponents.client5	httpclient5	5.2.1

Group ID	Artifact ID	Version
org.apache.httpcomponents.client5	httpclient5-cache	5.2.1
org.apache.httpcomponents.client5	httpclient5-fluent	5.2.1
org.apache.httpcomponents.client5	httpclient5-win	5.2.1
org.apache.httpcomponents.core5	httpcore5	5.2.3
org.apache.httpcomponents.core5	httpcore5-h2	5.2.3
org.apache.httpcomponents.core5	httpcore5-reactive	5.2.3
org.apache.kafka	connect	3.4.1
org.apache.kafka	connect-api	3.4.1
org.apache.kafka	connect-basic-auth-extension	3.4.1
org.apache.kafka	connect-file	3.4.1
org.apache.kafka	connect-json	3.4.1
org.apache.kafka	connect-mirror	3.4.1
org.apache.kafka	connect-mirror-client	3.4.1
org.apache.kafka	connect-runtime	3.4.1
org.apache.kafka	connect-transforms	3.4.1
org.apache.kafka	generator	3.4.1
org.apache.kafka	kafka-clients	3.4.1
org.apache.kafka	kafka-log4j-appender	3.4.1
org.apache.kafka	kafka-metadata	3.4.1
org.apache.kafka	kafka-raft	3.4.1
org.apache.kafka	kafka-server-common	3.4.1
org.apache.kafka	kafka-shell	3.4.1
org.apache.kafka	kafka-storage	3.4.1
org.apache.kafka	kafka-storage-api	3.4.1
org.apache.kafka	kafka-streams	3.4.1
org.apache.kafka	kafka-streams-scala_2.12	3.4.1
org.apache.kafka	kafka-streams-scala_2.13	3.4.1
org.apache.kafka	kafka-streams-test-utils	3.4.1
org.apache.kafka	kafka-tools	3.4.1
org.apache.kafka	kafka_2.12	3.4.1
org.apache.kafka	kafka_2.13	3.4.1
org.apache.kafka	trogdor	3.4.1
org.apache.logging.log4j	log4j-1.2-api	2.20.0
org.apache.logging.log4j	log4j-api	2.20.0

Group ID	Artifact ID	Version
org.apache.logging.log4j	log4j-api-test	2.20.0
org.apache.logging.log4j	log4j-appserver	2.20.0
org.apache.logging.log4j	log4j-cassandra	2.20.0
org.apache.logging.log4j	log4j-core	2.20.0
org.apache.logging.log4j	log4j-core-test	2.20.0
org.apache.logging.log4j	log4j-couchdb	2.20.0
org.apache.logging.log4j	log4j-docker	2.20.0
org.apache.logging.log4j	log4j-flume-ng	2.20.0
org.apache.logging.log4j	log4j-iostreams	2.20.0
org.apache.logging.log4j	log4j-jakarta-smtp	2.20.0
org.apache.logging.log4j	log4j-jakarta-web	2.20.0
org.apache.logging.log4j	log4j-jcl	2.20.0
org.apache.logging.log4j	log4j-jmx-gui	2.20.0
org.apache.logging.log4j	log4j-jpa	2.20.0
org.apache.logging.log4j	log4j-jpl	2.20.0
org.apache.logging.log4j	log4j-jul	2.20.0
org.apache.logging.log4j	log4j-kubernetes	2.20.0
org.apache.logging.log4j	log4j-layout-template-json	2.20.0
org.apache.logging.log4j	log4j-layout-template-json-test	2.20.0
org.apache.logging.log4j	log4j-mongodb3	2.20.0
org.apache.logging.log4j	log4j-mongodb4	2.20.0
org.apache.logging.log4j	log4j-slf4j-impl	2.20.0
org.apache.logging.log4j	log4j-slf4j2-impl	2.20.0
org.apache.logging.log4j	log4j-spring-boot	2.20.0
org.apache.logging.log4j	log4j-spring-cloud-config-client	2.20.0
org.apache.logging.log4j	log4j-taglib	2.20.0
org.apache.logging.log4j	log4j-to-jul	2.20.0
org.apache.logging.log4j	log4j-to-slf4j	2.20.0
org.apache.logging.log4j	log4j-web	2.20.0
org.apache.maven.plugin-tools	maven-plugin-annotations	3.6.4
org.apache.tomcat	tomcat-annotations-api	10.1.16
org.apache.tomcat	tomcat-jdbc	10.1.16
org.apache.tomcat	tomcat-jsp-api	10.1.16
org.apache.tomcat.embed	tomcat-embed-core	10.1.16
org.apache.tomcat.embed	tomcat-embed-el	10.1.16
org.apache.tomcat.embed	tomcat-embed-jasper	10.1.16
org.apache.tomcat.embed	tomcat-embed-websocket	10.1.16

Group ID	Artifact ID	Version
org.aspectj	aspectjrt	1.9.20.1
org.aspectj	aspectjtools	1.9.20.1
org.aspectj	aspectjweaver	1.9.20.1
org.assertj	assertj-core	3.24.2
org.assertj	assertj-guava	3.24.2
org.awaitility	awaitility	4.2.0
org.awaitility	awaitility-groovy	4.2.0
org.awaitility	awaitility-kotlin	4.2.0
org.awaitility	awaitility-scala	4.2.0
org.cache2k	cache2k-api	2.6.1.Final
org.cache2k	cache2k-config	2.6.1.Final
org.cache2k	cache2k-core	2.6.1.Final
org.cache2k	cache2k-jcache	2.6.1.Final
org.cache2k	cache2k-micrometer	2.6.1.Final
org.cache2k	cache2k-spring	2.6.1.Final
org.codehaus.janino	commons-compiler	3.1.10
org.codehaus.janino	commons-compiler-jdk	3.1.10
org.codehaus.janino	janino	3.1.10
org.eclipse	yasson	3.0.3
org.eclipse.angus	angus-activation	2.0.1
org.eclipse.angus	angus-core	1.1.0
org.eclipse.angus	angus-mail	1.1.0
org.eclipse.angus	dsn	1.1.0
org.eclipse.angus	gimap	1.1.0
org.eclipse.angus	imap	1.1.0
org.eclipse.angus	jakarta.mail	1.1.0
org.eclipse.angus	logging-mailhandler	1.1.0
org.eclipse.angus	pop3	1.1.0
org.eclipse.angus	smtp	1.1.0
org.eclipse.jetty	apache-jsp	11.0.18
org.eclipse.jetty	glassfish-jstl	11.0.18
org.eclipse.jetty	infinispan-common	11.0.18
org.eclipse.jetty	infinispan-embedded-query	11.0.18
org.eclipse.jetty	infinispan-remote-query	11.0.18
org.eclipse.jetty	jetty-alpn-client	11.0.18
org.eclipse.jetty	jetty-alpn-conscrypt-client	11.0.18
org.eclipse.jetty	jetty-alpn-conscrypt-server	11.0.18
org.eclipse.jetty	jetty-alpn-java-client	11.0.18

Group ID	Artifact ID	Version
org.eclipse.jetty	jetty-alpn-java-server	11.0.18
org.eclipse.jetty	jetty-alpn-server	11.0.18
org.eclipse.jetty	jetty-annotations	11.0.18
org.eclipse.jetty	jetty-ant	11.0.18
org.eclipse.jetty	jetty-cdi	11.0.18
org.eclipse.jetty	jetty-client	11.0.18
org.eclipse.jetty	jetty-deploy	11.0.18
org.eclipse.jetty	jetty-hazelcast	11.0.18
org.eclipse.jetty	jetty-home	11.0.18
org.eclipse.jetty	jetty-http	11.0.18
org.eclipse.jetty	jetty-http-spi	11.0.18
org.eclipse.jetty	jetty-io	11.0.18
org.eclipse.jetty	jetty-jaas	11.0.18
org.eclipse.jetty	jetty-jaspi	11.0.18
org.eclipse.jetty	jetty-jmx	11.0.18
org.eclipse.jetty	jetty-jndi	11.0.18
org.eclipse.jetty	jetty-keystore	11.0.18
org.eclipse.jetty	jetty-nosql	11.0.18
org.eclipse.jetty	jetty-openid	11.0.18
org.eclipse.jetty	jetty-plus	11.0.18
org.eclipse.jetty	jetty-proxy	11.0.18
org.eclipse.jetty	jetty-quickstart	11.0.18
org.eclipse.jetty	jetty-reactive-httpclient	3.0.10
org.eclipse.jetty	jetty-rewrite	11.0.18
org.eclipse.jetty	jetty-security	11.0.18
org.eclipse.jetty	jetty-server	11.0.18
org.eclipse.jetty	jetty-servlet	11.0.18
org.eclipse.jetty	jetty-servlets	11.0.18
org.eclipse.jetty	jetty-slf4j-impl	11.0.18
org.eclipse.jetty	jetty-unixdomain-server	11.0.18
org.eclipse.jetty	jetty-unixsocket-client	11.0.18
org.eclipse.jetty	jetty-unixsocket-common	11.0.18
org.eclipse.jetty	jetty-unixsocket-server	11.0.18
org.eclipse.jetty	jetty-util	11.0.18
org.eclipse.jetty	jetty-util-ajax	11.0.18
org.eclipse.jetty	jetty-webapp	11.0.18
org.eclipse.jetty	jetty-xml	11.0.18
org.eclipse.jetty.fcgi	fcgi-client	11.0.18

Group ID	Artifact ID	Version
org.eclipse.jetty.fcgi	fcgi-server	11.0.18
org.eclipse.jetty.gcloud	jetty-gcloud-session-manager	11.0.18
org.eclipse.jetty.http2	http2-client	11.0.18
org.eclipse.jetty.http2	http2-common	11.0.18
org.eclipse.jetty.http2	http2-hpack	11.0.18
org.eclipse.jetty.http2	http2-http-client-transport	11.0.18
org.eclipse.jetty.http2	http2-server	11.0.18
org.eclipse.jetty.http3	http3-client	11.0.18
org.eclipse.jetty.http3	http3-common	11.0.18
org.eclipse.jetty.http3	http3-http-client-transport	11.0.18
org.eclipse.jetty.http3	http3-qpack	11.0.18
org.eclipse.jetty.http3	http3-server	11.0.18
org.eclipse.jetty.memcached	jetty-memcached-sessions	11.0.18
org.eclipse.jetty.osgi	jetty-httpservice	11.0.18
org.eclipse.jetty.osgi	jetty-osgi-alpn	11.0.18
org.eclipse.jetty.osgi	jetty-osgi-boot	11.0.18
org.eclipse.jetty.osgi	jetty-osgi-boot-jsp	11.0.18
org.eclipse.jetty.osgi	jetty-osgi-boot-warurl	11.0.18
org.eclipse.jetty.quic	quic-client	11.0.18
org.eclipse.jetty.quic	quic-common	11.0.18
org.eclipse.jetty.quic	quic-quiche-common	11.0.18
org.eclipse.jetty.quic	quic-quiche-jna	11.0.18
org.eclipse.jetty.quic	quic-server	11.0.18
org.eclipse.jetty.websocket	websocket-core-client	11.0.18
org.eclipse.jetty.websocket	websocket-core-common	11.0.18
org.eclipse.jetty.websocket	websocket-core-server	11.0.18
org.eclipse.jetty.websocket	websocket-jakarta-client	11.0.18
org.eclipse.jetty.websocket	websocket-jakarta-common	11.0.18
org.eclipse.jetty.websocket	websocket-jakarta-server	11.0.18
org.eclipse.jetty.websocket	websocket-jetty-api	11.0.18
org.eclipse.jetty.websocket	websocket-jetty-client	11.0.18
org.eclipse.jetty.websocket	websocket-jetty-common	11.0.18
org.eclipse.jetty.websocket	websocket-jetty-server	11.0.18
org.eclipse.jetty.websocket	websocket-servlet	11.0.18
org.ehcache	ehcache	3.10.8
org.ehcache	ehcache-clustered	3.10.8
org.ehcache	ehcache-transactions	3.10.8
org.elasticsearch.client	elasticsearch-rest-client	8.7.1

Group ID	Artifact ID	Version
org.elasticsearch.client	elasticsearch-rest-client-sniffer	8.7.1
org.firebirdsql.jdbc	jaybird	5.0.2.java11
org.flywaydb	flyway-core	9.16.3
org.flywaydb	flyway-firebird	9.16.3
org.flywaydb	flyway-mysql	9.16.3
org.flywaydb	flyway-sqlserver	9.16.3
org.freemarker	freemarker	2.3.32
org.glassfish.jaxb	codemodel	4.0.4
org.glassfish.jaxb	jaxb-core	4.0.4
org.glassfish.jaxb	jaxb-jxc	4.0.4
org.glassfish.jaxb	jaxb-runtime	4.0.4
org.glassfish.jaxb	jaxb-xjc	4.0.4
org.glassfish.jaxb	txw2	4.0.4
org.glassfish.jaxb	xsom	4.0.4
org.glassfish.jersey.bundles	jaxrs-ri	3.1.3
org.glassfish.jersey.connectors	jersey-apache-connector	3.1.3
org.glassfish.jersey.connectors	jersey-apache5-connector	3.1.3
org.glassfish.jersey.connectors	jersey-grizzly-connector	3.1.3
org.glassfish.jersey.connectors	jersey-helidon-connector	3.1.3
org.glassfish.jersey.connectors	jersey-jdk-connector	3.1.3
org.glassfish.jersey.connectors	jersey-jetty-connector	3.1.3
org.glassfish.jersey.connectors	jersey-jnh-connector	3.1.3
org.glassfish.jersey.connectors	jersey-netty-connector	3.1.3
org.glassfish.jersey.containers	jersey-container-grizzly2-http	3.1.3
org.glassfish.jersey.containers	jersey-container-grizzly2-servlet	3.1.3
org.glassfish.jersey.containers	jersey-container-jdk-http	3.1.3
org.glassfish.jersey.containers	jersey-container-jetty-http	3.1.3
org.glassfish.jersey.containers	jersey-container-jetty-servlet	3.1.3

Group ID	Artifact ID	Version
org.glassfish.jersey.containers	jersey-container-netty-http	3.1.3
org.glassfish.jersey.containers	jersey-container-servlet	3.1.3
org.glassfish.jersey.containers	jersey-container-servlet-core	3.1.3
org.glassfish.jersey.containers	jersey-container-simple-http	3.1.3
org.glassfish.jersey.containers.glassfish	jersey-gf-ejb	3.1.3
org.glassfish.jersey.core	jersey-client	3.1.3
org.glassfish.jersey.core	jersey-common	3.1.3
org.glassfish.jersey.core	jersey-server	3.1.3
org.glassfish.jersey.ext	jersey-bean-validation	3.1.3
org.glassfish.jersey.ext	jersey-declarative-linking	3.1.3
org.glassfish.jersey.ext	jersey-entity-filtering	3.1.3
org.glassfish.jersey.ext	jersey-metainf-services	3.1.3
org.glassfish.jersey.ext	jersey-mvc	3.1.3
org.glassfish.jersey.ext	jersey-mvc-bean-validation	3.1.3
org.glassfish.jersey.ext	jersey-mvc-freemarker	3.1.3
org.glassfish.jersey.ext	jersey-mvc-jsp	3.1.3
org.glassfish.jersey.ext	jersey-mvc-mustache	3.1.3
org.glassfish.jersey.ext	jersey-proxy-client	3.1.3
org.glassfish.jersey.ext	jersey-spring6	3.1.3
org.glassfish.jersey.ext	jersey-wadl-doclet	3.1.3
org.glassfish.jersey.ext.cdi	jersey-cdi-rs-inject	3.1.3
org.glassfish.jersey.ext.cdi	jersey-cdi1x	3.1.3
org.glassfish.jersey.ext.cdi	jersey-cdi1x-ban-custom-hk2-binding	3.1.3
org.glassfish.jersey.ext.cdi	jersey-cdi1x-servlet	3.1.3
org.glassfish.jersey.ext.cdi	jersey-cdi1x-transaction	3.1.3
org.glassfish.jersey.ext.cdi	jersey-cdi1x-validation	3.1.3
org.glassfish.jersey.ext.cdi	jersey-weld2-se	3.1.3
org.glassfish.jersey.ext.microprofile	jersey-mp-config	3.1.3
org.glassfish.jersey.ext.microprofile	jersey-mp-rest-client	3.1.3
org.glassfish.jersey.ext.rx	jersey-rx-client-guava	3.1.3
org.glassfish.jersey.ext.rx	jersey-rx-client-rxjava	3.1.3
org.glassfish.jersey.ext.rx	jersey-rx-client-rxjava2	3.1.3
org.glassfish.jersey.inject	jersey-cdi2-se	3.1.3

Group ID	Artifact ID	Version
org.glassfish.jersey.inject	jersey-hk2	3.1.3
org.glassfish.jersey.media	jersey-media-jaxb	3.1.3
org.glassfish.jersey.media	jersey-media-json-binding	3.1.3
org.glassfish.jersey.media	jersey-media-json-gson	3.1.3
org.glassfish.jersey.media	jersey-media-json-jackson	3.1.3
org.glassfish.jersey.media	jersey-media-json-jettison	3.1.3
org.glassfish.jersey.media	jersey-media-json-processing	3.1.3
org.glassfish.jersey.media	jersey-media-kryo	3.1.3
org.glassfish.jersey.media	jersey-media-moxy	3.1.3
org.glassfish.jersey.media	jersey-media-multipart	3.1.3
org.glassfish.jersey.media	jersey-media-sse	3.1.3
org.glassfish.jersey.security	oauth1-client	3.1.3
org.glassfish.jersey.security	oauth1-server	3.1.3
org.glassfish.jersey.security	oauth1-signature	3.1.3
org.glassfish.jersey.security	oauth2-client	3.1.3
org.glassfish.jersey.test-framework	jersey-test-framework-core	3.1.3
org.glassfish.jersey.test-framework	jersey-test-framework-util	3.1.3
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-bundle	3.1.3
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-external	3.1.3
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-grizzly2	3.1.3
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-inmemory	3.1.3
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-jdk-http	3.1.3
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-jetty	3.1.3
org.glassfish.jersey.test-framework.providers	jersey-test-framework-provider-simple	3.1.3
org.glassfish.web	jakarta.servlet.jsp.jstl	3.0.1
org.hamcrest	hamcrest	2.2
org.hamcrest	hamcrest-core	2.2
org.hamcrest	hamcrest-library	2.2
org.hibernate.orm	hibernate-agroal	6.2.13.Final
org.hibernate.orm	hibernate-ant	6.2.13.Final
org.hibernate.orm	hibernate-c3p0	6.2.13.Final
org.hibernate.orm	hibernate-community-dialects	6.2.13.Final

Group ID	Artifact ID	Version
org.hibernate.orm	hibernate-core	6.2.13.Final
org.hibernate.orm	hibernate-envers	6.2.13.Final
org.hibernate.orm	hibernate-graalvm	6.2.13.Final
org.hibernate.orm	hibernate-hikaricp	6.2.13.Final
org.hibernate.orm	hibernate-jcache	6.2.13.Final
org.hibernate.orm	hibernate-jpamodelgen	6.2.13.Final
org.hibernate.orm	hibernate-micrometer	6.2.13.Final
org.hibernate.orm	hibernate-proxool	6.2.13.Final
org.hibernate.orm	hibernate-spatial	6.2.13.Final
org.hibernate.orm	hibernate-testing	6.2.13.Final
org.hibernate.orm	hibernate-vibur	6.2.13.Final
org.hibernate.validator	hibernate-validator	8.0.1.Final
org.hibernate.validator	hibernate-validator-annotation-processor	8.0.1.Final
org.hsqldb	hsqldb	2.7.2
org.infinispan	infinispan-anchored-keys	14.0.21.Final
org.infinispan	infinispan-api	14.0.21.Final
org.infinispan	infinispan-cachestore-jdbc	14.0.21.Final
org.infinispan	infinispan-cachestore-jdbc-common	14.0.21.Final
org.infinispan	infinispan-cachestore-jdbc-common-jakarta	14.0.21.Final
org.infinispan	infinispan-cachestore-jdbc-jakarta	14.0.21.Final
org.infinispan	infinispan-cachestore-remote	14.0.21.Final
org.infinispan	infinispan-cachestore-rocksdb	14.0.21.Final
org.infinispan	infinispan-cachestore-sql	14.0.21.Final
org.infinispan	infinispan-cdi-common	14.0.21.Final
org.infinispan	infinispan-cdi-common-jakarta	14.0.21.Final
org.infinispan	infinispan-cdi-embedded	14.0.21.Final
org.infinispan	infinispan-cdi-embedded-jakarta	14.0.21.Final
org.infinispan	infinispan-cdi-remote	14.0.21.Final
org.infinispan	infinispan-cdi-remote-jakarta	14.0.21.Final
org.infinispan	infinispan-checkstyle	14.0.21.Final
org.infinispan	infinispan-cli-client	14.0.21.Final
org.infinispan	infinispan-cli-client-jakarta	14.0.21.Final
org.infinispan	infinispan-client-hotrod	14.0.21.Final
org.infinispan	infinispan-client-hotrod-jakarta	14.0.21.Final

Group ID	Artifact ID	Version
org.infinispan	infinispan-client-rest	14.0.21.Final
org.infinispan	infinispan-client-rest-jakarta	14.0.21.Final
org.infinispan	infinispan-cloudevents-integration	14.0.21.Final
org.infinispan	infinispan-clustered-counter	14.0.21.Final
org.infinispan	infinispan-clustered-lock	14.0.21.Final
org.infinispan	infinispan-commons	14.0.21.Final
org.infinispan	infinispan-commons-jakarta	14.0.21.Final
org.infinispan	infinispan-commons-test	14.0.21.Final
org.infinispan	infinispan-component-annotations	14.0.21.Final
org.infinispan	infinispan-component-processor	14.0.21.Final
org.infinispan	infinispan-console	14.0.13.Final
org.infinispan	infinispan-core	14.0.21.Final
org.infinispan	infinispan-core-jakarta	14.0.21.Final
org.infinispan	infinispan-extended-statistics	14.0.21.Final
org.infinispan	infinispan-hibernate-cache-commons	14.0.21.Final
org.infinispan	infinispan-hibernate-cache-spi	14.0.21.Final
org.infinispan	infinispan-hibernate-cache-v60	14.0.21.Final
org.infinispan	infinispan-hibernate-cache-v62	14.0.21.Final
org.infinispan	infinispan-hotrod	14.0.21.Final
org.infinispan	infinispan-hotrod-jakarta	14.0.21.Final
org.infinispan	infinispan-jboss-marshalling	14.0.21.Final
org.infinispan	infinispan-jcache	14.0.21.Final
org.infinispan	infinispan-jcache-commons	14.0.21.Final
org.infinispan	infinispan-jcache-remote	14.0.21.Final
org.infinispan	infinispan-key-value-store-client	14.0.21.Final
org.infinispan	infinispan-marshaller-kryo	14.0.21.Final
org.infinispan	infinispan-marshaller-kryo-bundle	14.0.21.Final
org.infinispan	infinispan-marshaller-protostuff	14.0.21.Final
org.infinispan	infinispan-marshaller-protostuff-bundle	14.0.21.Final
org.infinispan	infinispan-multimap	14.0.21.Final
org.infinispan	infinispan-multimap-jakarta	14.0.21.Final
org.infinispan	infinispan-objectfilter	14.0.21.Final
org.infinispan	infinispan-query	14.0.21.Final
org.infinispan	infinispan-query-core	14.0.21.Final

Group ID	Artifact ID	Version
org.infinispan	infinispan-query-dsl	14.0.21.Final
org.infinispan	infinispan-query-jakarta	14.0.21.Final
org.infinispan	infinispan-remote-query-client	14.0.21.Final
org.infinispan	infinispan-remote-query-server	14.0.21.Final
org.infinispan	infinispan-scripting	14.0.21.Final
org.infinispan	infinispan-server-core	14.0.21.Final
org.infinispan	infinispan-server-hotrod	14.0.21.Final
org.infinispan	infinispan-server-hotrod-jakarta	14.0.21.Final
org.infinispan	infinispan-server-memcached	14.0.21.Final
org.infinispan	infinispan-server-resp	14.0.21.Final
org.infinispan	infinispan-server-rest	14.0.21.Final
org.infinispan	infinispan-server-router	14.0.21.Final
org.infinispan	infinispan-server-runtime	14.0.21.Final
org.infinispan	infinispan-server-testdriver-core	14.0.21.Final
org.infinispan	infinispan-server-testdriver-core-jakarta	14.0.21.Final
org.infinispan	infinispan-server-testdriver-junit4	14.0.21.Final
org.infinispan	infinispan-server-testdriver-junit5	14.0.21.Final
org.infinispan	infinispan-spring-boot-starter-embedded	14.0.21.Final
org.infinispan	infinispan-spring-boot-starter-remote	14.0.21.Final
org.infinispan	infinispan-spring-boot3-starter-embedded	14.0.21.Final
org.infinispan	infinispan-spring-boot3-starter-remote	14.0.21.Final
org.infinispan	infinispan-spring5-common	14.0.21.Final
org.infinispan	infinispan-spring5-embedded	14.0.21.Final
org.infinispan	infinispan-spring5-remote	14.0.21.Final
org.infinispan	infinispan-spring6-common	14.0.21.Final
org.infinispan	infinispan-spring6-embedded	14.0.21.Final
org.infinispan	infinispan-spring6-remote	14.0.21.Final
org.infinispan	infinispan-tasks	14.0.21.Final
org.infinispan	infinispan-tasks-api	14.0.21.Final
org.infinispan	infinispan-tools	14.0.21.Final
org.infinispan	infinispan-tools-jakarta	14.0.21.Final
org.infinispan.protostream	protostream	4.6.5.Final

Group ID	Artifact ID	Version
org.infinispan.protostream	protostream-processor	4.6.5.Final
org.infinispan.protostream	protostream-types	4.6.5.Final
org.influxdb	influxdb-java	2.23
org.jboss.logging	jboss-logging	3.5.3.Final
org.jdom	jdom2	2.0.6.1
org.jetbrains.kotlin	kotlin-compiler	1.8.22
org.jetbrains.kotlin	kotlin-compiler-embeddable	1.8.22
org.jetbrains.kotlin	kotlin-daemon-client	1.8.22
org.jetbrains.kotlin	kotlin-main-kts	1.8.22
org.jetbrains.kotlin	kotlin-osgi-bundle	1.8.22
org.jetbrains.kotlin	kotlin-reflect	1.8.22
org.jetbrains.kotlin	kotlin-script-runtime	1.8.22
org.jetbrains.kotlin	kotlin-script-util	1.8.22
org.jetbrains.kotlin	kotlin-scripting-common	1.8.22
org.jetbrains.kotlin	kotlin-scripting-ide-services	1.8.22
org.jetbrains.kotlin	kotlin-scripting-jvm	1.8.22
org.jetbrains.kotlin	kotlin-scripting-jvm-host	1.8.22
org.jetbrains.kotlin	kotlin-stdlib	1.8.22
org.jetbrains.kotlin	kotlin-stdlib-common	1.8.22
org.jetbrains.kotlin	kotlin-stdlib-jdk7	1.8.22
org.jetbrains.kotlin	kotlin-stdlib-jdk8	1.8.22
org.jetbrains.kotlin	kotlin-stdlib-js	1.8.22
org.jetbrains.kotlin	kotlin-test	1.8.22
org.jetbrains.kotlin	kotlin-test-annotations-common	1.8.22
org.jetbrains.kotlin	kotlin-test-common	1.8.22
org.jetbrains.kotlin	kotlin-test-js	1.8.22
org.jetbrains.kotlin	kotlin-test-junit	1.8.22
org.jetbrains.kotlin	kotlin-test-junit5	1.8.22
org.jetbrains.kotlin	kotlin-test-testng	1.8.22
org.jetbrains.kotlinx	kotlinx-coroutines-android	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-core	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-core-jvm	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-debug	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-guava	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-javafx	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-jdk8	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-jdk9	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-play-services	1.6.4

Group ID	Artifact ID	Version
org.jetbrains.kotlinx	kotlinx-coroutines-reactive	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-reactor	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-rx2	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-rx3	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-slf4j	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-swing	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-test	1.6.4
org.jetbrains.kotlinx	kotlinx-coroutines-test-jvm	1.6.4
org.jooq	jooq	3.18.7
org.jooq	jooq-codegen	3.18.7
org.jooq	jooq-kotlin	3.18.7
org.jooq	jooq-meta	3.18.7
org.junit.jupiter	junit-jupiter	5.9.3
org.junit.jupiter	junit-jupiter-api	5.9.3
org.junit.jupiter	junit-jupiter-engine	5.9.3
org.junit.jupiter	junit-jupiter-migrationsupport	5.9.3
org.junit.jupiter	junit-jupiter-params	5.9.3
org.junit.platform	junit-platform-commons	1.9.3
org.junit.platform	junit-platform-console	1.9.3
org.junit.platform	junit-platform-engine	1.9.3
org.junit.platform	junit-platform-jfr	1.9.3
org.junit.platform	junit-platform-launcher	1.9.3
org.junit.platform	junit-platform-reporting	1.9.3
org.junit.platform	junit-platform-runner	1.9.3
org.junit.platform	junit-platform-suite	1.9.3
org.junit.platform	junit-platform-suite-api	1.9.3
org.junit.platform	junit-platform-suite-commons	1.9.3
org.junit.platform	junit-platform-suite-engine	1.9.3
org.junit.platform	junit-platform-testkit	1.9.3
org.junit.vintage	junit-vintage-engine	5.9.3
org.jvnet.staxex	stax-ex	2.1.0
org.liquibase	liquibase-cdi	4.20.0
org.liquibase	liquibase-core	4.20.0
org.mariadb	r2dbc-mariadb	1.1.4
org.mariadb.jdbc	mariadb-java-client	3.1.4
org.messaginghub	pooled-jms	3.1.5
org.mockito	mockito-android	5.3.1
org.mockito	mockito-core	5.3.1

Group ID	Artifact ID	Version
org.mockito	mockito-errorprone	5.3.1
org.mockito	mockito-junit-jupiter	5.3.1
org.mockito	mockito-proxy	5.3.1
org.mockito	mockito-subclass	5.3.1
org.mongodb	bson	4.9.1
org.mongodb	bson-record-codec	4.9.1
org.mongodb	mongodb-driver-core	4.9.1
org.mongodb	mongodb-driver-legacy	4.9.1
org.mongodb	mongodb-driver-reactivestreams	4.9.1
org.mongodb	mongodb-driver-sync	4.9.1
org.neo4j.driver	neo4j-java-driver	5.13.0
org.postgresql	postgresql	42.6.0
org.postgresql	r2dbc-postgresql	1.0.2.RELEASE
org.projectlombok	lombok	1.18.30
org.quartz-scheduler	quartz	2.3.2
org.quartz-scheduler	quartz-jobs	2.3.2
org.reactivestreams	reactive-streams	1.0.4
org.seleniumhq.selenium	htmlunit-driver	4.8.3
org.seleniumhq.selenium	lift	4.8.3
org.seleniumhq.selenium	selenium-api	4.8.3
org.seleniumhq.selenium	selenium-chrome-driver	4.8.3
org.seleniumhq.selenium	selenium-chromium-driver	4.8.3
org.seleniumhq.selenium	selenium-devtools-v109	4.8.3
org.seleniumhq.selenium	selenium-devtools-v110	4.8.3
org.seleniumhq.selenium	selenium-devtools-v111	4.8.3
org.seleniumhq.selenium	selenium-devtools-v85	4.8.3
org.seleniumhq.selenium	selenium-edge-driver	4.8.3
org.seleniumhq.selenium	selenium-firefox-driver	4.8.3
org.seleniumhq.selenium	selenium-grid	4.8.3
org.seleniumhq.selenium	selenium-http	4.8.3
org.seleniumhq.selenium	selenium-http-jdk-client	4.8.3
org.seleniumhq.selenium	selenium-ie-driver	4.8.3
org.seleniumhq.selenium	selenium-java	4.8.3
org.seleniumhq.selenium	selenium-json	4.8.3
org.seleniumhq.selenium	selenium-manager	4.8.3
org.seleniumhq.selenium	selenium-remote-driver	4.8.3
org.seleniumhq.selenium	selenium-safari-driver	4.8.3
org.seleniumhq.selenium	selenium-session-map-jdbc	4.8.3

Group ID	Artifact ID	Version
org.seleniumhq.selenium	selenium-session-map-redis	4.8.3
org.seleniumhq.selenium	selenium-support	4.8.3
org.skyscreamer	jsonassert	1.5.1
org.slf4j	jcl-over-slf4j	2.0.9
org.slf4j	jul-to-slf4j	2.0.9
org.slf4j	log4j-over-slf4j	2.0.9
org.slf4j	slf4j-api	2.0.9
org.slf4j	slf4j-ext	2.0.9
org.slf4j	slf4j-jdk-platform-logging	2.0.9
org.slf4j	slf4j-jdk14	2.0.9
org.slf4j	slf4j-log4j12	2.0.9
org.slf4j	slf4j-nop	2.0.9
org.slf4j	slf4j-reload4j	2.0.9
org.slf4j	slf4j-simple	2.0.9
org.springframework	spring-aop	6.0.14
org.springframework	spring-aspects	6.0.14
org.springframework	spring-beans	6.0.14
org.springframework	spring-context	6.0.14
org.springframework	spring-context-indexer	6.0.14
org.springframework	spring-context-support	6.0.14
org.springframework	spring-core	6.0.14
org.springframework	spring-core-test	6.0.14
org.springframework	spring-expression	6.0.14
org.springframework	spring-instrument	6.0.14
org.springframework	spring-jcl	6.0.14
org.springframework	spring-jdbc	6.0.14
org.springframework	spring-jms	6.0.14
org.springframework	spring-messaging	6.0.14
org.springframework	spring-orm	6.0.14
org.springframework	spring-oxm	6.0.14
org.springframework	spring-r2dbc	6.0.14
org.springframework	spring-test	6.0.14
org.springframework	spring-tx	6.0.14
org.springframework	spring-web	6.0.14
org.springframework	spring-webflux	6.0.14
org.springframework	spring-webmvc	6.0.14
org.springframework	spring-websocket	6.0.14
org.springframework.amqp	spring-amqp	3.0.10

Group ID	Artifact ID	Version
org.springframework.amqp	spring-rabbit	3.0.10
org.springframework.amqp	spring-rabbit-junit	3.0.10
org.springframework.amqp	spring-rabbit-stream	3.0.10
org.springframework.amqp	spring-rabbit-test	3.0.10
org.springframework.batch	spring-batch-core	5.0.4
org.springframework.batch	spring-batch-infrastructure	5.0.4
org.springframework.batch	spring-batch-integration	5.0.4
org.springframework.batch	spring-batch-test	5.0.4
org.springframework.boot	spring-boot	3.1.6
org.springframework.boot	spring-boot-actuator	3.1.6
org.springframework.boot	spring-boot-actuator-autoconfigure	3.1.6
org.springframework.boot	spring-boot-autoconfigure	3.1.6
org.springframework.boot	spring-boot-autoconfigure-processor	3.1.6
org.springframework.boot	spring-boot-buildpack-platform	3.1.6
org.springframework.boot	spring-boot-configuration-metadata	3.1.6
org.springframework.boot	spring-boot-configuration-processor	3.1.6
org.springframework.boot	spring-boot-devtools	3.1.6
org.springframework.boot	spring-boot-docker-compose	3.1.6
org.springframework.boot	spring-boot-jarmode-layertools	3.1.6
org.springframework.boot	spring-boot-loader	3.1.6
org.springframework.boot	spring-boot-loader-tools	3.1.6
org.springframework.boot	spring-boot-properties-migrator	3.1.6
org.springframework.boot	spring-boot-starter	3.1.6
org.springframework.boot	spring-boot-starter-activemq	3.1.6
org.springframework.boot	spring-boot-starter-actuator	3.1.6
org.springframework.boot	spring-boot-starter-amqp	3.1.6
org.springframework.boot	spring-boot-starter-aop	3.1.6
org.springframework.boot	spring-boot-starter-artemis	3.1.6
org.springframework.boot	spring-boot-starter-batch	3.1.6
org.springframework.boot	spring-boot-starter-cache	3.1.6
org.springframework.boot	spring-boot-starter-data-cassandra	3.1.6
org.springframework.boot	spring-boot-starter-data-cassandra-reactive	3.1.6
org.springframework.boot	spring-boot-starter-data-couchbase	3.1.6

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-data-couchbase-reactive	3.1.6
org.springframework.boot	spring-boot-starter-data-elasticsearch	3.1.6
org.springframework.boot	spring-boot-starter-data-jdbc	3.1.6
org.springframework.boot	spring-boot-starter-data-jpa	3.1.6
org.springframework.boot	spring-boot-starter-data-ldap	3.1.6
org.springframework.boot	spring-boot-starter-data-mongodb	3.1.6
org.springframework.boot	spring-boot-starter-data-mongodb-reactive	3.1.6
org.springframework.boot	spring-boot-starter-data-neo4j	3.1.6
org.springframework.boot	spring-boot-starter-data-r2dbc	3.1.6
org.springframework.boot	spring-boot-starter-data-redis	3.1.6
org.springframework.boot	spring-boot-starter-data-redis-reactive	3.1.6
org.springframework.boot	spring-boot-starter-data-rest	3.1.6
org.springframework.boot	spring-boot-starter-freemarker	3.1.6
org.springframework.boot	spring-boot-starter-graphql	3.1.6
org.springframework.boot	spring-boot-starter-groovy-templates	3.1.6
org.springframework.boot	spring-boot-starter-hateoas	3.1.6
org.springframework.boot	spring-boot-starter-integration	3.1.6
org.springframework.boot	spring-boot-starter-jdbc	3.1.6
org.springframework.boot	spring-boot-starter-jersey	3.1.6
org.springframework.boot	spring-boot-starter-jetty	3.1.6
org.springframework.boot	spring-boot-starter-jooq	3.1.6
org.springframework.boot	spring-boot-starter-json	3.1.6
org.springframework.boot	spring-boot-starter-log4j2	3.1.6
org.springframework.boot	spring-boot-starter-logging	3.1.6
org.springframework.boot	spring-boot-starter-mail	3.1.6
org.springframework.boot	spring-boot-starter-mustache	3.1.6
org.springframework.boot	spring-boot-starter-oauth2-authorization-server	3.1.6
org.springframework.boot	spring-boot-starter-oauth2-client	3.1.6
org.springframework.boot	spring-boot-starter-oauth2-resource-server	3.1.6
org.springframework.boot	spring-boot-starter-quartz	3.1.6
org.springframework.boot	spring-boot-starter-reactor-netty	3.1.6

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-rsocket	3.1.6
org.springframework.boot	spring-boot-starter-security	3.1.6
org.springframework.boot	spring-boot-starter-test	3.1.6
org.springframework.boot	spring-boot-starter-thymeleaf	3.1.6
org.springframework.boot	spring-boot-starter-tomcat	3.1.6
org.springframework.boot	spring-boot-starter-undertow	3.1.6
org.springframework.boot	spring-boot-starter-validation	3.1.6
org.springframework.boot	spring-boot-starter-web	3.1.6
org.springframework.boot	spring-boot-starter-web-services	3.1.6
org.springframework.boot	spring-boot-starter-webflux	3.1.6
org.springframework.boot	spring-boot-starter-websocket	3.1.6
org.springframework.boot	spring-boot-test	3.1.6
org.springframework.boot	spring-boot-test-autoconfigure	3.1.6
org.springframework.boot	spring-boot-testcontainers	3.1.6
org.springframework.data	spring-data-cassandra	4.1.6
org.springframework.data	spring-data-commons	3.1.6
org.springframework.data	spring-data-couchbase	5.1.6
org.springframework.data	spring-data-elasticsearch	5.1.6
org.springframework.data	spring-data-envers	3.1.6
org.springframework.data	spring-data-jdbc	3.1.6
org.springframework.data	spring-data-jpa	3.1.6
org.springframework.data	spring-data-keyvalue	3.1.6
org.springframework.data	spring-data-ldap	3.1.6
org.springframework.data	spring-data-mongodb	4.1.6
org.springframework.data	spring-data-neo4j	7.1.6
org.springframework.data	spring-data-r2dbc	3.1.6
org.springframework.data	spring-data-redis	3.1.6
org.springframework.data	spring-data-relational	3.1.6
org.springframework.data	spring-data-rest-core	4.1.6
org.springframework.data	spring-data-rest-hal-explorer	4.1.6
org.springframework.data	spring-data-rest-webmvc	4.1.6
org.springframework.graphql	spring-graphql	1.2.4
org.springframework.graphql	spring-graphql-test	1.2.4
org.springframework.hateoas	spring-hateoas	2.1.3
org.springframework.integration	spring-integration-amqp	6.1.5
org.springframework.integration	spring-integration-camel	6.1.5

Group ID	Artifact ID	Version
org.springframework.integration	spring-integration-cassandra	6.1.5
org.springframework.integration	spring-integration-core	6.1.5
org.springframework.integration	spring-integration-event	6.1.5
org.springframework.integration	spring-integration-feed	6.1.5
org.springframework.integration	spring-integration-file	6.1.5
org.springframework.integration	spring-integration-ftp	6.1.5
org.springframework.integration	spring-integration-graphql	6.1.5
org.springframework.integration	spring-integration-groovy	6.1.5
org.springframework.integration	spring-integration-hazelcast	6.1.5
org.springframework.integration	spring-integration-http	6.1.5
org.springframework.integration	spring-integration-ip	6.1.5
org.springframework.integration	spring-integration-jdbc	6.1.5
org.springframework.integration	spring-integration-jms	6.1.5
org.springframework.integration	spring-integration-jmx	6.1.5
org.springframework.integration	spring-integration-jpa	6.1.5
org.springframework.integration	spring-integration-kafka	6.1.5
org.springframework.integration	spring-integration-mail	6.1.5
org.springframework.integration	spring-integration-mongodb	6.1.5
org.springframework.integration	spring-integration-mqtt	6.1.5
org.springframework.integration	spring-integration-r2dbc	6.1.5
org.springframework.integration	spring-integration-redis	6.1.5
org.springframework.integration	spring-integration-rsocket	6.1.5
org.springframework.integration	spring-integration-scripting	6.1.5

Group ID	Artifact ID	Version
org.springframework.integration	spring-integration-security	6.1.5
org.springframework.integration	spring-integration-sftp	6.1.5
org.springframework.integration	spring-integration-smb	6.1.5
org.springframework.integration	spring-integration-stomp	6.1.5
org.springframework.integration	spring-integration-stream	6.1.5
org.springframework.integration	spring-integration-syslog	6.1.5
org.springframework.integration	spring-integration-test	6.1.5
org.springframework.integration	spring-integration-test-support	6.1.5
org.springframework.integration	spring-integration-webflux	6.1.5
org.springframework.integration	spring-integration-websocket	6.1.5
org.springframework.integration	spring-integration-ws	6.1.5
org.springframework.integration	spring-integration-xml	6.1.5
org.springframework.integration	spring-integration-xmpp	6.1.5
org.springframework.integration	spring-integration-zeromq	6.1.5
org.springframework.integration	spring-integration-zip	6.1.5
org.springframework.integration	spring-integration-zookeeper	6.1.5
org.springframework.kafka	spring-kafka	3.0.13
org.springframework.kafka	spring-kafka-test	3.0.13
org.springframework.ldap	spring-ldap-core	3.1.2
org.springframework.ldap	spring-ldap-ldif-core	3.1.2
org.springframework.ldap	spring-ldap-odm	3.1.2
org.springframework.ldap	spring-ldap-test	3.1.2
org.springframework.restdocs	spring-restdocs-asciidoctor	3.0.1
org.springframework.restdocs	spring-restdocs-core	3.0.1
org.springframework.restdocs	spring-restdocs-mockmvc	3.0.1
org.springframework.restdocs	spring-restdocs-restassured	3.0.1
org.springframework.restdocs	spring-restdocs-webtestclient	3.0.1

Group ID	Artifact ID	Version
org.springframework.retry	spring-retry	2.0.4
org.springframework.security	spring-security-acl	6.1.5
org.springframework.security	spring-security-aspects	6.1.5
org.springframework.security	spring-security-cas	6.1.5
org.springframework.security	spring-security-config	6.1.5
org.springframework.security	spring-security-core	6.1.5
org.springframework.security	spring-security-crypto	6.1.5
org.springframework.security	spring-security-data	6.1.5
org.springframework.security	spring-security-ldap	6.1.5
org.springframework.security	spring-security-messaging	6.1.5
org.springframework.security	spring-security-oauth2-authorization-server	1.1.3
org.springframework.security	spring-security-oauth2-client	6.1.5
org.springframework.security	spring-security-oauth2-core	6.1.5
org.springframework.security	spring-security-oauth2-jose	6.1.5
org.springframework.security	spring-security-oauth2-resource-server	6.1.5
org.springframework.security	spring-security-rsocket	6.1.5
org.springframework.security	spring-security-saml2-service-provider	6.1.5
org.springframework.security	spring-security-taglibs	6.1.5
org.springframework.security	spring-security-test	6.1.5
org.springframework.security	spring-security-web	6.1.5
org.springframework.session	spring-session-core	3.1.3
org.springframework.session	spring-session-data-mongodb	3.1.3
org.springframework.session	spring-session-data-redis	3.1.3
org.springframework.session	spring-session-hazelcast	3.1.3
org.springframework.session	spring-session-jdbc	3.1.3
org.springframework.ws	spring-ws-core	4.0.8
org.springframework.ws	spring-ws-security	4.0.8
org.springframework.ws	spring-ws-support	4.0.8
org.springframework.ws	spring-ws-test	4.0.8
org.springframework.ws	spring-xml	4.0.8
org.testcontainers	azure	1.18.3
org.testcontainers	cassandra	1.18.3
org.testcontainers	clickhouse	1.18.3
org.testcontainers	cockroachdb	1.18.3
org.testcontainers	consul	1.18.3
org.testcontainers	couchbase	1.18.3

Group ID	Artifact ID	Version
org.testcontainers	cratedb	1.18.3
org.testcontainers	database-commons	1.18.3
org.testcontainers	db2	1.18.3
org.testcontainers	dynalite	1.18.3
org.testcontainers	elasticsearch	1.18.3
org.testcontainers	gcloud	1.18.3
org.testcontainers	hivemq	1.18.3
org.testcontainers	influxdb	1.18.3
org.testcontainers	jdbc	1.18.3
org.testcontainers	junit-jupiter	1.18.3
org.testcontainers	k3s	1.18.3
org.testcontainers	kafka	1.18.3
org.testcontainers	localstack	1.18.3
org.testcontainers	mariadb	1.18.3
org.testcontainers	mockserver	1.18.3
org.testcontainers	mongodb	1.18.3
org.testcontainers	mssqlserver	1.18.3
org.testcontainers	mysql	1.18.3
org.testcontainers	neo4j	1.18.3
org.testcontainers	nginx	1.18.3
org.testcontainers	oracle-xe	1.18.3
org.testcontainers	orientdb	1.18.3
org.testcontainers	postgresql	1.18.3
org.testcontainers	presto	1.18.3
org.testcontainers	pulsar	1.18.3
org.testcontainers	questdb	1.18.3
org.testcontainers	r2dbc	1.18.3
org.testcontainers	rabbitmq	1.18.3
org.testcontainers	redpanda	1.18.3
org.testcontainers	selenium	1.18.3
org.testcontainers	solace	1.18.3
org.testcontainers	solr	1.18.3
org.testcontainers	spock	1.18.3
org.testcontainers	testcontainers	1.18.3
org.testcontainers	tidb	1.18.3
org.testcontainers	toxiproxy	1.18.3
org.testcontainers	trino	1.18.3
org.testcontainers	vault	1.18.3

Group ID	Artifact ID	Version
org.testcontainers	yugabytedb	1.18.3
org.thymeleaf	thymeleaf	3.1.2.RELEASE
org.thymeleaf	thymeleaf-spring6	3.1.2.RELEASE
org.thymeleaf.extras	thymeleaf-extras-springsecurity6	3.1.2.RELEASE
org.webjars	webjars-locator-core	0.52
org.xerial	sqlite-jdbc	3.41.2.2
org.xmlunit	xmlunit-assertj	2.9.1
org.xmlunit	xmlunit-assertj3	2.9.1
org.xmlunit	xmlunit-core	2.9.1
org.xmlunit	xmlunit-jakarta-jaxb-impl	2.9.1
org.xmlunit	xmlunit-legacy	2.9.1
org.xmlunit	xmlunit-matchers	2.9.1
org.xmlunit	xmlunit-placeholders	2.9.1
org.yaml	snakeyaml	1.33
redis.clients	jedis	4.3.2
wsdl4j	wsdl4j	1.6.3

.F.2. Version Properties

The following table provides all properties that can be used to override the versions managed by Spring Boot. Browse the spring-boot-dependencies build.gradle for a complete list of dependencies. You can learn how to customize these versions in your application in the Build Tool Plugins documentation.

Library	Version Property
ActiveMQ	activemq.version
Angus Mail	angus-mail.version
Artemis	artemis.version
AspectJ	aspectj.version
AssertJ	assertj.version
Awaitility	awaitility.version
Brave	brave.version
Build Helper Maven Plugin	build-helper-maven-plugin.version
Byte Buddy	byte-buddy.version
cache2k	cache2k.version
Caffeine	caffeine.version
Cassandra Driver	cassandra-driver.version
Classmate	classmate.version
Commons Codec	commons-codec.version

Library	Version Property
Commons DBCP2	commons-dbcp2.version
Commons Lang3	commons-lang3.version
Commons Pool	commons-pool.version
Commons Pool2	commons-pool2.version
Couchbase Client	couchbase-client.version
DB2 JDBC	db2-jdbc.version
Dependency Management Plugin	dependency-management-plugin.version
Derby	derby.version
Dropwizard Metrics	dropwizard-metrics.version
Ehcache3	ehcache3.version
Elasticsearch Client	elasticsearch-client.version
Flyway	flyway.version
FreeMarker	freemarker.version
Git Commit ID Maven Plugin	git-commit-id-maven-plugin.version
Glassfish JAXB	glassfish-jaxb.version
Glassfish JSTL	glassfish-jstl.version
GraphQL Java	graphql-java.version
Groovy	groovy.version
Gson	gson.version
H2	h2.version
Hamcrest	hamcrest.version
Hazelcast	hazelcast.version
Hibernate	hibernate.version
Hibernate Validator	hibernate-validator.version
HikariCP	hikaricp.version
HSQLDB	hsqldb.version
HtmlUnit	htmlunit.version
HttpAsyncClient	httpasyncclient.version
HttpClient5	httpclient5.version
HttpCore	httpcore.version
HttpCore5	httpcore5.version
Infinispan	infinispan.version
InfluxDB Java	influxdb-java.version
Jackson Bom	jackson-bom.version
Jakarta Activation	jakarta-activation.version
Jakarta Annotation	jakarta-annotation.version
Jakarta JMS	jakarta-jms.version
Jakarta Json	jakarta-json.version

Library	Version Property
Jakarta Json Bind	jakarta-json-bind.version
Jakarta Mail	jakarta-mail.version
Jakarta Management	jakarta-management.version
Jakarta Persistence	jakarta-persistence.version
Jakarta Servlet	jakarta-servlet.version
Jakarta Servlet JSP JSTL	jakarta-servlet-jsp-jstl.version
Jakarta Transaction	jakarta-transaction.version
Jakarta Validation	jakarta-validation.version
Jakarta WebSocket	jakarta-websocket.version
Jakarta WS RS	jakarta-ws-rs.version
Jakarta XML Bind	jakarta-xml-bind.version
Jakarta XML SOAP	jakarta-xml-soap.version
Jakarta XML WS	jakarta-xml-ws.version
Janino	janino.version
Javax Cache	javax-cache.version
Javax Money	javax-money.version
Jaxen	jaxen.version
Jaybird	jaybird.version
JBoss Logging	jboss-logging.version
JDOM2	jdom2.version
Jedis	jedis.version
Jersey	jersey.version
Jetty	jetty.version
Jetty Reactive HTTPClient	jetty-reactive-httpclient.version
JMustache	jmustache.version
jOOQ	jooq.version
Json Path	json-path.version
Json-smart	json-smart.version
JsonAssert	jsonassert.version
JTDS	jtds.version
JUnit	junit.version
JUnit Jupiter	junit-jupiter.version
Kafka	kafka.version
Kotlin	kotlin.version
Kotlin Coroutines	kotlin-coroutines.version
Lettuce	lettuce.version
Liquibase	liquibase.version
Log4j2	log4j2.version

Library	Version Property
Logback	logback.version
Lombok	lombok.version
MariaDB	mariadb.version
Maven AntRun Plugin	maven-antrun-plugin.version
Maven Assembly Plugin	maven-assembly-plugin.version
Maven Clean Plugin	maven-clean-plugin.version
Maven Compiler Plugin	maven-compiler-plugin.version
Maven Dependency Plugin	maven-dependency-plugin.version
Maven Deploy Plugin	maven-deploy-plugin.version
Maven Enforcer Plugin	maven-enforcer-plugin.version
Maven Failsafe Plugin	maven-failsafe-plugin.version
Maven Help Plugin	maven-help-plugin.version
Maven Install Plugin	maven-install-plugin.version
Maven Invoker Plugin	maven-invoker-plugin.version
Maven Jar Plugin	maven-jar-plugin.version
Maven Javadoc Plugin	maven-javadoc-plugin.version
Maven Resources Plugin	maven-resources-plugin.version
Maven Shade Plugin	maven-shade-plugin.version
Maven Source Plugin	maven-source-plugin.version
Maven Surefire Plugin	maven-surefire-plugin.version
Maven War Plugin	maven-war-plugin.version
Micrometer	micrometer.version
Micrometer Tracing	micrometer-tracing.version
Mockito	mockito.version
MongoDB	mongodb.version
MSSQL JDBC	mssql-jdbc.version
MySQL	mysql.version
Native Build Tools Plugin	native-build-tools-plugin.version
NekoHTML	nekohtml.version
Neo4j Java Driver	neo4j-java-driver.version
Netty	netty.version
OkHttp	okhttp.version
OpenTelemetry	opentelemetry.version
Oracle Database	oracle-database.version
Oracle R2DBC	oracle-r2dbc.version
Pooled JMS	pooled-jms.version
Postgresql	postgresql.version
Prometheus Client	prometheus-client.version

Library	Version Property
Quartz	quartz.version
QueryDSL	querydsl.version
R2DBC H2	r2dbc-h2.version
R2DBC MariaDB	r2dbc-mariadb.version
R2DBC MSSQL	r2dbc-mssql.version
R2DBC MySQL	r2dbc-mysql.version
R2DBC Pool	r2dbc-pool.version
R2DBC Postgresql	r2dbc-postgresql.version
R2DBC Proxy	r2dbc-proxy.version
R2DBC SPI	r2dbc-spi.version
Rabbit AMQP Client	rabbit-amqp-client.version
Rabbit Stream Client	rabbit-stream-client.version
Reactive Streams	reactive-streams.version
Reactor Bom	reactor-bom.version
REST Assured	rest-assured.version
RSocket	rsocket.version
RxJava3	rxjava3.version
SAAJ Impl	saaj-impl.version
Selenium	selenium.version
Selenium HtmlUnit	selenium-htmlunit.version
SendGrid	sendgrid.version
SLF4J	slf4j.version
SnakeYAML	snakeyaml.version
Spring AMQP	spring-amqp.version
Spring Authorization Server	spring-authorization-server.version
Spring Batch	spring-batch.version
Spring Data Bom	spring-data-bom.version
Spring Framework	spring-framework.version
Spring GraphQL	spring-graphql.version
Spring HATEOAS	spring-hateoas.version
Spring Integration	spring-integration.version
Spring Kafka	spring-kafka.version
Spring LDAP	spring-ldap.version
Spring RESTDocs	spring-restdocs.version
Spring Retry	spring-retry.version
Spring Security	spring-security.version
Spring Session	spring-session.version
Spring WS	spring-ws.version

Library	Version Property
SQLite JDBC	sqlite-jdbc.version
Testcontainers	testcontainers.version
Thymeleaf	thymeleaf.version
Thymeleaf Extras Data Attribute	thymeleaf-extras-data-attribute.version
Thymeleaf Extras SpringSecurity	thymeleaf-extras-springsecurity.version
Thymeleaf Layout Dialect	thymeleaf-layout-dialect.version
Tomcat	tomcat.version
UnboundID LDAPSDK	unboundid-ldapsdk.version
Undertow	undertow.version
Versions Maven Plugin	versions-maven-plugin.version
WebJars Locator Core	webjars-locator-core.version
WSDL4j	wsdl4j.version
XML Maven Plugin	xml-maven-plugin.version
XmlUnit2	xmlunit2.version
Yasson	yasson.version