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
Table of Contents

1. Spring Boot Documentation ................................................................. 2
   1.1. About the Documentation ......................................................... 2
   1.2. Getting Help ............................................................................. 2
   1.3. Upgrading from an Earlier Version ............................................ 2
   1.4. First Steps ................................................................................ 2
   1.5. Working with Spring Boot .......................................................... 2
   1.6. Learning about Spring Boot Features ......................................... 3
   1.7. Moving to Production ................................................................. 3
   1.8. Advanced Topics ....................................................................... 3
2. Getting Started ................................................................................... 5
   2.1. Introducing Spring Boot .............................................................. 5
   2.2. System Requirements ................................................................ 5
       2.2.1. Servlet Containers .............................................................. 5
   2.3. Installing Spring Boot ................................................................. 6
       2.3.1. Installation Instructions for the Java Developer ..................... 6
       Maven Installation ....................................................................... 6
       Gradle Installation ...................................................................... 7
       2.3.2. Installing the Spring Boot CLI ............................................ 7
       Manual Installation ..................................................................... 7
       Installation with SDKMAN! ......................................................... 7
       OSX Homebrew Installation ......................................................... 8
       MacPorts Installation .................................................................. 9
       Command-line Completion ........................................................ 9
       Windows Scoop Installation ......................................................... 9
       Quick-start Spring CLI Example .................................................. 10
       2.3.3. Upgrading from an Earlier Version of Spring Boot .............. 10
2.4. Developing Your First Spring Boot Application ............................... 11
   2.4.1. Creating the POM ................................................................. 11
   2.4.2. Adding Classpath Dependencies ........................................... 12
   2.4.3. Writing the Code .................................................................. 13
       The @RestController and @RequestMapping Annotations .......... 14
       The @EnableAutoConfiguration Annotation ............................... 14
       The “main” Method ................................................................... 15
   2.4.4. Running the Example ........................................................... 15
   2.4.5. Creating an Executable Jar .................................................... 15
2.5. What to Read Next ......................................................................... 17
3. Using Spring Boot ............................................................................. 19
4. Spring Boot Features

4.1. SpringApplication

4.1.1. Startup Failure

4.1.2. Lazy Initialization

4.1.3. Customizing the Banner

4.1.4. Customizing SpringApplication

4.1.5. Fluent Builder API

4.1.6. Application Availability

4.1.7. Application Events and Listeners

4.1.8. Web Environment

4.1.9. Accessing Application Arguments

4.1.10. Using the ApplicationRunner or CommandLineRunner

4.1.11. Application Exit

4.1.12. Admin Features

4.1.13. Application Startup tracking

4.2. Externalized Configuration

4.2.1. Accessing Command Line Properties

4.2.2. JSON Application Properties

4.2.3. External Application Properties

4.2.4. Encrypting Properties

4.2.5. Working with YAML

4.2.6. Configuring Random Values

4.2.7. Type-safe Configuration Properties

JavaBean properties binding

Constructor binding

Enabling @ConfigurationProperties-annotated types
4.12. Working with NoSQL Technologies

4.12.1. Redis
- Connecting to Redis

4.12.2. MongoDB
- Connecting to a MongoDB Database
- MongoTemplate
- Spring Data MongoDB Repositories
- Embedded Mongo

4.12.3. Neo4j
- Connecting to a Neo4j Database
- Spring Data Neo4j Repositories

4.12.4. Solr
- Connecting to Solr
- Spring Data Solr Repositories

4.12.5. Elasticsearch
- Connecting to Elasticsearch using REST clients
- Connecting to Elasticsearch using Reactive REST clients
- Connecting to Elasticsearch by Using Spring Data
- Spring Data Elasticsearch Repositories

4.12.6. Cassandra
- Connecting to Cassandra
- Spring Data Cassandra Repositories

4.12.7. Couchbase
9.11. Messaging

9.11.1. Disable Transacted JMS Session

9.12. Batch Applications

9.12.1. Specifying a Batch Data Source

9.12.2. Running Spring Batch Jobs on Startup

9.12.3. Running from the Command Line

9.12.4. Storing the Job Repository

9.13. Actuator

9.13.1. Change the HTTP Port or Address of the Actuator Endpoints

9.13.2. Customize the ‘whitelabel’ Error Page

9.13.3. Sanitize Sensitive Values

9.13.4. Map Health Indicators to Micrometer Metrics


9.14.2. Change the UserDetailsService and Add User Accounts

9.14.3. Enable HTTPS When Running behind a Proxy Server

9.15. Hot Swapping

9.15.1. Reload Static Content

9.15.2. Reload Templates without Restarting the Container

9.15.3. Fast Application Restarts

9.15.4. Reload Java Classes without Restarting the Container

9.16. Build

9.16.1. Generate Build Information

9.16.2. Generate Git Information

9.16.3. Customize Dependency Versions

9.16.4. Create an Executable JAR with Maven

9.16.5. Use a Spring Boot Application as a Dependency

9.16.6. Extract Specific Libraries When an Executable Jar Runs

9.16.7. Create a Non-executable JAR with Exclusions

9.16.8. Remote Debug a Spring Boot Application Started with Maven

9.16.9. Build an Executable Archive from Ant without Using spring-boot-antlib

9.17. Traditional Deployment

9.17.1. Create a Deployable War File

9.17.2. Convert an Existing Application to Spring Boot

9.17.3. Deploying a WAR to WebLogic

9.17.4. Use Jedis Instead of Lettuce

9.17.5. Use Testcontainers for integration testing
Appendix A: Common Application properties

10.A.1. Core Properties ................................................. 417
10.A.2. Cache Properties ............................................. 426
10.A.3. Mail Properties .............................................. 428
10.A.4. JSON Properties .............................................. 428
10.A.7. Data Migration Properties .................................. 465
10.A.8. Integration Properties ....................................... 472
10.A.10. Templating Properties ..................................... 500
10.A.13. RSocket Properties ........................................ 521
10.A.15. Devtools Properties ....................................... 549

Appendix B: Configuration Metadata ..................................... 551

10.B.1. Metadata Format .............................................. 551
  Group Attributes .................................................. 553
  Property Attributes .............................................. 554
  Hint Attributes ................................................... 556
  Repeated Metadata Items ......................................... 557
  Value Hint .......................................................... 558
  Value Providers .................................................... 558
10.B.3. Generating Your Own Metadata by Using the Annotation Processor .......................................................... 565
  Configuring the Annotation Processor .......................... 565
  Automatic Metadata Generation ................................. 566
  Adding Additional Metadata ...................................... 569

Appendix C: Auto-configuration Classes .................................. 569

10.C.1. spring-boot-autoconfigure .................................. 570
10.C.2. spring-boot-actuator-autoconfigure ....................... 574

Appendix D: Test Auto-configuration Annotations ............................ 577

10.D.1. Test Slices ...................................................... 577

Appendix E: The Executable Jar Format ................................... 584

10.E.1. Nested JARs ..................................................... 584
  The Executable Jar File Structure ................................. 584
  The Executable War File Structure ............................... 585
  Index Files ............................................................ 586
  Classpath Index ...................................................... 586
Legal

2.4.2

Copyright © 2012-2021

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 1. Spring Boot Documentation

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

1.1. About the Documentation

The Spring Boot reference guide is available as:

- Multi-page HTML
- Single page HTML
- PDF

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

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.

1.2. 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.

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.

1.3. Upgrading from an Earlier Version

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.

You should always ensure that you are running a supported version of Spring Boot.
1.4. 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

1.5. Working 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

1.6. Learning about Spring Boot Features

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

- **Core Features**: SpringApplication | External Configuration | Profiles | Logging
- **Web Applications**: MVC | Embedded Containers
- **Working with data**: SQL | NO-SQL
- **Messaging**: Overview | JMS
- **Testing**: Overview | Boot Applications | Utils
- **Extending**: Auto-configuration | @Conditions

1.7. 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 Tracing | Process

1.8. 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 2. 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.

2.1. 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. We also provide a command line tool that runs “spring scripts”.

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 and no requirement for XML configuration.

2.2. System Requirements

Spring Boot 2.4.2 requires Java 8 and is compatible up to Java 15 (included). Spring Framework 5.3.3 or above is also required.

Explicit build support is provided for the following build tools:

<table>
<thead>
<tr>
<th>Build Tool</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maven</td>
<td>3.3+</td>
</tr>
<tr>
<td>Gradle</td>
<td>6 (6.3 or later). 5.6.x is also supported but in a deprecated form</td>
</tr>
</tbody>
</table>

2.2.1. Servlet Containers

Spring Boot supports the following embedded servlet containers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Servlet Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomcat 9.0</td>
<td>4.0</td>
</tr>
<tr>
<td>Name</td>
<td>Servlet Version</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Jetty 9.4</td>
<td>3.1</td>
</tr>
<tr>
<td>Undertow 2.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

You can also deploy Spring Boot applications to any Servlet 3.1+ compatible container.

# 2.3. 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 v1.8 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.

## 2.3.1. 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 3.3 or above. If you do not already have Maven installed, you can follow the instructions at maven.apache.org.

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` groupId. 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 6 (6.3 or later). Gradle 5.6.x is also supported but this support is deprecated and will be removed in a future release. 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.

2.3.2. 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. It lets you run Groovy scripts, which means that you have a familiar Java-like syntax without so much boilerplate code.

You do not need to use the CLI to work with Spring Boot, but it is definitely the quickest way to get a Spring application off the ground.

Manual Installation

You can download the Spring CLI distribution from the Spring software repository:

- spring-boot-cli-2.4.2-bin.zip
- spring-boot-cli-2.4.2-bin.tar.gz

Cutting edge snapshot distributions are also available.

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

```bash
$ sdk install springboot
$ spring --version
Spring Boot v2.4.2
```

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:

```bash
$ sdk ls springboot
```

```
================================================================================
Available Springboot Versions
================================================================================
> + dev
* 2.4.2
```

### OSX Homebrew Installation

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

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

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

```bash
$ 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 /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:

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

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:

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

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

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.

Quick-start Spring CLI Example

You can use the following web application to test your installation. To start, create a file called app.groovy, as follows:
```java
@RestController
class ThisWillActuallyRun {

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

}
```

Then run it from a shell, as follows:

```
$ spring run app.groovy
```

![info]

The first run of your application is slow, as dependencies are downloaded. Subsequent runs are much quicker.

Open `localhost:8080` in your favorite web browser. You should see the following output:

```
Hello World!
```

### 2.3.3. Upgrading from an Earlier Version of Spring Boot

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.

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.

![info]

Once you’re done with the migration, please make sure to remove this module from your project’s dependencies.
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.

## 2.4. 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. We use Maven to build this project, since most IDEs support it.

The [spring.io](http://spring.io) web site contains many “Getting Started” guides that use Spring Boot. If you need to solve a specific problem, check there first.

You can shortcut the steps below by going to [start.spring.io](http://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 [Spring Initializr documentation](http://spring.io) for more details.

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

```
$ java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)
```

```
$ mvn -v
Apache Maven 3.5.4 (1eddde0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06-17T14:33:14-04:00)
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java version: 1.8.0_102, vendor: Oracle Corporation
```

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.

### 2.4.1. Creating the POM

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:
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).

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.

### 2.4.2. Adding Classpath Dependencies

Spring Boot provides a number of “Starters” that let you add jars to your classpath. Our applications for smoke tests 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.

Other “Starters” provide dependencies that you are likely to need when developing a specific type
of application. 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:

```xml
<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.

### 2.4.3. Writing the Code

To finish our application, we need to create a single Java file. By default, Maven compiles sources from `src/main/java`, so you need to create that directory structure and then add a file named `src/main/java/Example.java` to contain the following code:
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 `Example` 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.

The `@RestController` and `@RequestMapping` annotations are Spring MVC annotations (they are not specific to Spring Boot). See the [MVC section](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html) in the Spring Reference Documentation for more details.

**The @EnableAutoConfiguration Annotation**

The second class-level annotation is `@EnableAutoConfiguration`. This annotation 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 Example.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.

2.4.4. Running the Example

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

.......... . . . (log output here)
.......... . . .
.......... Started Example in 2.222 seconds (JVM 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.

2.4.5. 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.

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:

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

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:
If you look in the target directory, you should see myproject-0.0.1-SNAPSHOT.jar. The file should be around 10 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
```

As before, to exit the application, press ctrl-c.
your own applications. If you are a task-oriented type of developer, you might want to jump over to spring.io and check out 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 Using Spring Boot. If you are really impatient, you could also jump ahead and read about Spring Boot features.
Chapter 3. Using 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.

3.1. 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.

### 3.1.1. 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.

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.

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

### 3.1.2. Maven

To learn about using Spring Boot with Maven, please refer to the documentation for Spring Boot’s Maven plugin:

- Reference ([HTML](#) and [PDF](#))
- API

### 3.1.3. Gradle

To learn about using Spring Boot with Gradle, please refer to the documentation for Spring Boot’s Gradle plugin:
3.1.4. 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:

```xml
<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="2.4.2" />

  <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>

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”.

### 3.1.5. 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’s 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 STS 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring-boot-starter</code></td>
<td>Core starter, including auto-configuration support, logging and YAML</td>
</tr>
<tr>
<td><code>spring-boot-starter-activemq</code></td>
<td>Starter for JMS messaging using Apache ActiveMQ</td>
</tr>
<tr>
<td><code>spring-boot-starter-amqp</code></td>
<td>Starter for using Spring AMQP and Rabbit MQ</td>
</tr>
<tr>
<td><code>spring-boot-starter-aop</code></td>
<td>Starter for aspect-oriented programming with Spring AOP and AspectJ</td>
</tr>
<tr>
<td><code>spring-boot-starter-artemis</code></td>
<td>Starter for JMS messaging using Apache Artemis</td>
</tr>
<tr>
<td><code>spring-boot-starter-batch</code></td>
<td>Starter for using Spring Batch</td>
</tr>
<tr>
<td><code>spring-boot-starter-cache</code></td>
<td>Starter for using Spring Framework’s caching support</td>
</tr>
<tr>
<td><code>spring-boot-starter-data-cassandra</code></td>
<td>Starter for using Cassandra distributed database and Spring Data Cassandra</td>
</tr>
<tr>
<td><code>spring-boot-starter-data-cassandra-reactive</code></td>
<td>Starter for using Cassandra distributed database and Spring Data Cassandra Reactive</td>
</tr>
<tr>
<td><code>spring-boot-starter-data-couchbase</code></td>
<td>Starter for using Couchbase document-oriented database and Spring Data Couchbase</td>
</tr>
<tr>
<td><code>spring-boot-starter-data-couchbase-reactive</code></td>
<td>Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive</td>
</tr>
<tr>
<td><code>spring-boot-starter-data-elasticsearch</code></td>
<td>Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch</td>
</tr>
<tr>
<td><code>spring-boot-starter-data-jdbc</code></td>
<td>Starter for using Spring Data JDBC</td>
</tr>
<tr>
<td><code>spring-boot-starter-data-jpa</code></td>
<td>Starter for using Spring Data JPA</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring-boot-starter-data-ldap</td>
<td>Starter for using Spring Data LDAP</td>
</tr>
<tr>
<td>spring-boot-starter-data-mongodb</td>
<td>Starter for using MongoDB document-oriented database and Spring Data MongoDB</td>
</tr>
<tr>
<td>spring-boot-starter-data-mongodb-reactive</td>
<td>Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive</td>
</tr>
<tr>
<td>spring-boot-starter-data-neo4j</td>
<td>Starter for using Neo4j graph database and Spring Data Neo4j</td>
</tr>
<tr>
<td>spring-boot-starter-data-r2dbc</td>
<td>Starter for using Spring Data R2DBC</td>
</tr>
<tr>
<td>spring-boot-starter-data-redis</td>
<td>Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client</td>
</tr>
<tr>
<td>spring-boot-starter-data-redis-reactive</td>
<td>Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client</td>
</tr>
<tr>
<td>spring-boot-starter-data-rest</td>
<td>Starter for exposing Spring Data repositories over REST using Spring Data REST</td>
</tr>
<tr>
<td>spring-boot-starter-data-solr</td>
<td>Starter for using the Apache Solr search platform with Spring Data Solr</td>
</tr>
<tr>
<td>spring-boot-starter-freemarker</td>
<td>Starter for building MVC web applications using FreeMarker views</td>
</tr>
<tr>
<td>spring-boot-starter-groovy-templates</td>
<td>Starter for building MVC web applications using Groovy Templates views</td>
</tr>
<tr>
<td>spring-boot-starter-hateoas</td>
<td>Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS</td>
</tr>
<tr>
<td>spring-boot-starter-integration</td>
<td>Starter for using Spring Integration</td>
</tr>
<tr>
<td>spring-boot-starter-jdbc</td>
<td>Starter for using JDBC with the HikariCP connection pool</td>
</tr>
<tr>
<td>spring-boot-starter-jersey</td>
<td>Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web</td>
</tr>
<tr>
<td>spring-boot-starter-jooq</td>
<td>Starter for using jOOQ to access SQL databases. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc</td>
</tr>
<tr>
<td>spring-boot-starter-json</td>
<td>Starter for reading and writing json</td>
</tr>
<tr>
<td>spring-boot-starter-jta-atomikos</td>
<td>Starter for JTA transactions using Atomikos</td>
</tr>
<tr>
<td>spring-boot-starter-jta-bitronix</td>
<td>Starter for JTA transactions using Bitronix. Deprecated since 2.3.0</td>
</tr>
<tr>
<td>spring-boot-starter-mail</td>
<td>Starter for using Java Mail and Spring Framework's email sending support</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring-boot-starter-mustache</td>
<td>Starter for building web applications using Mustache views</td>
</tr>
<tr>
<td>spring-boot-starter-oauth2-client</td>
<td>Starter for using Spring Security's OAuth2/OpenID Connect client features</td>
</tr>
<tr>
<td>spring-boot-starter-oauth2-resource-server</td>
<td>Starter for using Spring Security's OAuth2 resource server features</td>
</tr>
<tr>
<td>spring-boot-starter-quartz</td>
<td>Starter for using the Quartz scheduler</td>
</tr>
<tr>
<td>spring-boot-starter-rsocket</td>
<td>Starter for building RSocket clients and servers</td>
</tr>
<tr>
<td>spring-boot-starter-security</td>
<td>Starter for using Spring Security</td>
</tr>
<tr>
<td>spring-boot-starter-test</td>
<td>Starter for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest and Mockito</td>
</tr>
<tr>
<td>spring-boot-starter-thymeleaf</td>
<td>Starter for building MVC web applications using Thymeleaf views</td>
</tr>
<tr>
<td>spring-boot-starter-validation</td>
<td>Starter for using Java Bean Validation with Hibernate Validator</td>
</tr>
<tr>
<td>spring-boot-starter-web</td>
<td>Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container</td>
</tr>
<tr>
<td>spring-boot-starter-web-services</td>
<td>Starter for using Spring Web Services</td>
</tr>
<tr>
<td>spring-boot-starter-webflux</td>
<td>Starter for building WebFlux applications using Spring Framework's Reactive Web support</td>
</tr>
<tr>
<td>spring-boot-starter-websocket</td>
<td>Starter for building WebSocket applications using Spring Framework's WebSocket support</td>
</tr>
</tbody>
</table>

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

**Table 2. Spring Boot production starters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring-boot-starter-actuator</td>
<td>Starter for using Spring Boot’s Actuator which provides production ready features to help you monitor and manage your application</td>
</tr>
</tbody>
</table>

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**
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring-boot-starter-jetty</td>
<td>Starter for using Jetty as the embedded servlet container. An alternative to spring-boot-starter-tomcat</td>
</tr>
<tr>
<td>spring-boot-starter-log4j2</td>
<td>Starter for using Log4j2 for logging. An alternative to spring-boot-starter-logging</td>
</tr>
<tr>
<td>spring-boot-starter-logging</td>
<td>Starter for logging using Logback. Default logging starter</td>
</tr>
<tr>
<td>spring-boot-starter-reactor-netty</td>
<td>Starter for using Reactor Netty as the embedded reactive HTTP server.</td>
</tr>
<tr>
<td>spring-boot-starter-tomcat</td>
<td>Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web</td>
</tr>
<tr>
<td>spring-boot-starter-undertow</td>
<td>Starter for using Undertow as the embedded servlet container. An alternative to spring-boot-starter-tomcat</td>
</tr>
</tbody>
</table>

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

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

3.2. Structuring Your Code

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

3.2.1. 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.

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

3.2.2. 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.

💡 If you don't want to use `@SpringBootApplication`, the `@EnableAutoConfiguration` and `@ComponentScan` annotations that it imports defines that behaviour so you can also use those instead.

The following listing shows a typical layout:

```java
package com.example.myapplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

3.3. 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`. 
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.

### 3.3.1. 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.

### 3.3.2. 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.

### 3.4. 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.

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.

### 3.4.1. 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.

### 3.4.2. 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:
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.

You can define exclusions both at the annotation level and by using the property.

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.

3.5. 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 often find that using @ComponentScan (to find your beans) and using @Autowired (to do constructor injection) works well.

If you structure your code as suggested above (locating your application class in a root package), you can add @ComponentScan without any arguments. All of your application components (@Component, @Service, @Repository, @Controller etc.) are automatically registered as Spring Beans.

The following example shows a @Service Bean that uses constructor injection to obtain a required RiskAssessor bean:
package com.example.service;

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

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    @Autowired
    public DatabaseAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
    }

    // ...
}

If a bean has one constructor, you can omit the @Autowired, as shown in the following example:

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...
}

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

3.6. 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)
• `@Configuration`: allow to register extra beans in the context or import additional configuration classes

```java
package com.example.myapplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

`@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
package com.example.myapplication;
import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
@Configuration(proxyBeanMethods = false)
@EnableAutoConfiguration
@Import({ MyConfig.class, MyAnotherConfig.class })
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

In this example, `Application` 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`).
3.7. 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.

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

3.7.1. 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.

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

3.7.2. 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
```

3.7.3. 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
```

### 3.7.4. 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
```

### 3.7.5. 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 Developer Tools section later in this chapter and the Hot swapping “How-to” for details.

### 3.8. 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>
```
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.

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.

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.

### 3.8.1. 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.

Because you need more information about web requests while developing Spring MVC and Spring WebFlux applications, developer tools will 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, etc. 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.
If you don't want property defaults to be applied you can set `spring.devtools.add-properties` to `false` in your `application.properties`.

For a complete list of the properties that are applied by the devtools, see `DevToolsPropertyDefaultsPostProcessor`.

### 3.8.2. 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. The way in which you cause the classpath to be updated depends on the IDE 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.

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.

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.

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)`).

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.
**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*

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

*Yaml*

```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*

```properties
spring.devtools.restart.exclude=static/**,public/**
```
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
public static void main(String[] args) {
    System.setProperty("spring.devtools.restart.enabled", "false");
    SpringApplication.run(MyApp.class, 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.

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:
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.

⚠️ 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. For most applications, this approach works well. However, it can sometimes cause classloading issues.

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. If you work on a multi-module project, and not every module is imported into your IDE, you may need to customize things. To do so, you can create 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-\.]\+[\w\d-\.]\+\.jar
restart.include.projectcommon=/mycorp-myproj-[^\w\d-\.]\+[\w\d-\.]\+\.jar

Yaml

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

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

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.

3.8.3. 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 from [livereload.com](http://livereload.com).

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`.

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.

To trigger LiveReload when a file changes, Automatic Restart must be enabled.

3.8.4. 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 file 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"
```

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.

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

```yml
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.

### 3.8.5. 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:

```xml
<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.
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 STS 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:

```
2015-06-10 18:25:07.074  INFO 14938 --- [           main] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
```

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

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.

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.

**3.9. 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 endpoints, consider adding `spring-boot-actuator`. See Spring Boot Actuator: Production-ready Features for details.

**3.10. 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 4. Spring Boot 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 "Using Spring Boot" sections, so that you have a good grounding of the basics.

4.1. 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
public static void main(String[] args) {
    SpringApplication.run(MySpringConfiguration.class, args);
}
```

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

```
2019-04-31 13:09:54.117  INFO 56603 --- [           main]
o.s.b.s.app.SampleApplication            : Starting SampleApplication v0.1.0 on
mycomputer with PID 56603 (/apps/myapp.jar started by pwebb)
2019-04-31 13:09:54.166  INFO 56603 --- [           main]
ationConfigServletWebServerApplicationContext : Refreshing
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@6e5a8246: startup date [Wed Jul 31 00:08:16 PDT 2013]; root of context hierarchy
2019-04-01 13:09:56.912  INFO 41370 --- [           main]
.o.s.b.s.app.SampleApplication            : Started SampleApplication in 2.992 seconds
(JVM running for 3.658)
```

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

### 4.1.1. 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's listening on port 8080 or configure this application to listen on another port.
```

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
```

### 4.1.2. 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
```

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.

### 4.1.3. 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`. In addition to a text file, you can also add a `banner.gif`, `banner.jpg`, or `banner.png` image file to your classpath or set the `spring.banner.image.location` property. Images are converted into an ASCII art representation and printed above any text banner.

Inside your `banner.txt` file, you can use any of the following placeholders:

**Table 4. Banner variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>${application.version}</code></td>
<td>The version number of your application, as declared in <code>MANIFEST.MF</code>. For example, <code>Implementation-Version: 1.0</code> is printed as <code>1.0</code>.</td>
</tr>
<tr>
<td><code>${application.formatted-version}</code></td>
<td>The version number of your application, as declared in <code>MANIFEST.MF</code> and formatted for display (surrounded with brackets and prefixed with <code>v</code>). For example <code>(v1.0)</code>.</td>
</tr>
</tbody>
</table>
### Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>${spring-boot.version}</code></td>
<td>The Spring Boot version that you are using. For example 2.4.2.</td>
</tr>
<tr>
<td><code>${spring-boot.formatted-version}</code></td>
<td>The Spring Boot version that you are using, formatted for display (surrounded with brackets and prefixed with v). For example (v2.4.2).</td>
</tr>
<tr>
<td><code>${Ansi.NAME}</code> (or <code>${AnsiColor.NAME}</code>, <code>${AnsiBackground.NAME}</code>, <code>${AnsiStyle.NAME}</code>)</td>
<td>Where NAME is the name of an ANSI escape code. See AnsiPropertySource for details.</td>
</tr>
<tr>
<td><code>${application.title}</code></td>
<td>The title of your application, as declared in MANIFEST.MF. For example Implementation-Title: MyApp is printed as MyApp.</td>
</tr>
</tbody>
</table>

The `SpringApplicationBuilder` 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`.

### 4.1.4. 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
public static void main(String[] args) {
    SpringApplication app = new SpringApplication(MySpringConfiguration.class);
    app.setBannerMode(Banner.Mode.OFF);
    app.run(args);
}
```

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 references to XML configuration or to packages that should be scanned.

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 `SpringApplicationBuilder` Javadoc.

### 4.1.5. 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);
```

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.

### 4.1.6. 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's 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.

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’s too busy for additional traffic.

An application is considered ready as soon as application and command-line runners have been
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
@Component
public class ReadinessStateExporter {

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

We can also update the state of the application, when the application breaks and cannot recover:
@Component
public class LocalCacheVerifier {
    private final ApplicationEventPublisher eventPublisher;

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

    public void checkLocalCache() {
        try {
            //...
        }
        catch (CacheCompletelyBrokenException ex) {
            AvailabilityChangeEvent.publish(this.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.

4.1.7. Application Events and Listeners

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

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:

```java
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 `SpringApplicationEvents` 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.

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.

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`. 
4.1.8. 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 `setApplicationContextClass(…)`.

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

4.1.9. 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 org.springframework.boot.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.stereotype.*;

@Component
public class MyBean {

    @Autowired
    public MyBean(ApplicationArguments args) {
        boolean debug = args.containsOption("debug");
        List<String> files = args.getNonOptionArgs();
        // if run with "--debug logfile.txt" debug=true, files=["logfile.txt"]
    }
}
```

Spring Boot also registers a `CommandLinePropertySource` with the Spring `Environment`. This lets you also inject single application arguments by using the `@Value` annotation.
4.1.10. 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.

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.*;
import org.springframework.stereotype.*;

@Component
public class MyBean implements CommandLineRunner {

    public void run(String... args) {
        // 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.

4.1.11. 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:
@SpringBootApplication
public class ExitCodeApplication {

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

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

4.1.12. 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.

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

4.1.13. 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 StartupSteps. 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:

    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(MySpringConfiguration.class);
        app.setApplicationStartup(new BufferingApplicationStartup(2048));
        app.run(args);
    }
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. Additionally, Spring Boot Actuator will expose a startup endpoint to expose this information as a JSON document.

### 4.2. 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, include 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. Properties are considered in the following order (with values from lower items overriding earlier ones):

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. @TestPropertySource annotations on your tests.

14. 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).

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

@Component
public class MyBean {

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

    // ...
}
```

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").

### 4.2.1. 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).
4.2.2. 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='{"acme":{"name":"test"}}' java -jar myapp.jar
```

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

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

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

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

```
$ java -jar myapp.jar --spring.application.json='{"acme":{"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`.

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.

4.2.3. 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. The classpath root
2. The classpath /config package
3. The current directory
4. The /config subdirectory in the current directory
5. 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. You can also refer to an explicit location by using the `spring.config.location` environment property (which is a comma-separated list of directory locations or file paths). The following example shows how to specify a different file name:

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

The following example shows how to specify two locations:

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

Use the prefix `optional:` if the locations are optional and you don't mind if they don't exist.

`spring.config.name` and `spring.config.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 used as-is. Whether specified directly or contained in a directory, configuration files must include a file extension in their name. Typical extensions that are supported out-of-the-box are `.properties`, `.yaml`, and `.yml`.

When multiple locations are specified, the later ones can override the values of earlier ones.

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 addition 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 the set of locations considered is:

1. `optional:classpath:/`
2. `optional:classpath:/config/`
3. `optional:file://`
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 don’t mind if it doesn’t 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. The 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.

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.

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.yml` and `application-prod.yml` 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 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.

Properties files are only ever loaded once. If you've already directly imported a profile specific property files then it won't 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

```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 doesn’t matter. For instance, the two examples below produce the same result:

**Properties**

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

**Yaml**

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

**Properties**

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

**Yaml**

```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.
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’s 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**

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

**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’s 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 use:

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.

⚠️ 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.

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.

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

**Property Placeholders**

The values in `application.properties` and `application.yml` are filtered through the existing `Environment` when they are used, so you can refer back to previously defined values (for example, from System properties). The standard `${name}` property-placeholder syntax can be used anywhere within a value.

For example, the following file will set `app.description` to “MyApp is a Spring Boot application”:

**Properties**

```
app.name=MyApp
app.description=${app.name} is a Spring Boot application
```

**Yaml**

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

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.yml` 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.config.activate.on-cloud-platform: kubernetes
spring.application.name: MyCloudApp
```

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

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

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 comments.

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.

**Activation Properties**

It’s sometimes useful to only activate a given get 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>
<thead>
<tr>
<th>Property</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on-profile</code></td>
<td>A profile expression that must match for the document to be active.</td>
</tr>
<tr>
<td><code>on-cloud-platform</code></td>
<td>The CloudPlatform that must be detected for the document to be active.</td>
</tr>
</tbody>
</table>

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

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

Yaml

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

### 4.2.4. 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're looking for a secure way to store credentials and passwords, the [Spring Cloud Vault](#) project provides support for storing externalized configuration in [HashiCorp Vault](#).

### 4.2.5. 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.

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

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.

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.

YAML Shortcomings

YAML files cannot be loaded by using the @PropertySource annotation. So, in the case that you need to load values that way, you need to use a properties file.
Using the multi-document YAML syntax in profile-specific YAML files can lead to unexpected behavior. For example, consider the following config in a file:

```
application-dev.yml
```

```
server.port: 8000
---
  spring.config.activate.on-profile: "!test"
  mypassword: "secret"
```

If you run the application with the argument `--spring.profiles.active=dev` you might expect `mypassword` to be set to “secret”, but this is not the case.

The nested document will be filtered because the main file is named `application-dev.yml`. It is already considered to be profile-specific, and nested documents will be ignored.

We recommend that you don’t mix profile-specific YAML files and multiple YAML documents. Stick to using only one of them.

### 4.2.6. 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).
4.2.7. 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.



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:

68


package com.example;

import java.net.InetAddress;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("acme")
public class AcmeProperties {

    private boolean enabled;
    private InetAddress remoteAddress;
    private final Security security = new Security();

    public boolean isEnabled() { ... }
    public void setEnabled(boolean enabled) { ... }
    public InetAddress getRemoteAddress() { ... }
    public void setRemoteAddress(InetAddress remoteAddress) { ... }
    public Security getSecurity() { ... }

    public static class Security {
        private String username;
        private String password;
        private List<String> roles = new ArrayList<>(Collections.singleton("USER"));

        public String getUsername() { ... }
        public void setUsername(String username) { ... }
        public String getPassword() { ... }
        public void setPassword(String password) { ... }
        public List<String> getRoles() { ... }
        public void setRoles(List<String> roles) { ... }
    }
}
The preceding POJO defines the following properties:

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

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

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
package com.example;

import java.net.InetAddress;
import java.util.List;
```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.ConstructorBinding;
import org.springframework.boot.context.properties.bind.DefaultValue;

@ConstructorBinding
@ConfigurationProperties("acme")
public class AcmeProperties {

    private final boolean enabled;
    private final InetAddress remoteAddress;
    private final Security security;

    public AcmeProperties(boolean enabled, InetAddress remoteAddress, Security security) {
        this.enabled = enabled;
        this.remoteAddress = remoteAddress;
        this.security = security;
    }

    public boolean isEnabled() { ... }
    public InetAddress getRemoteAddress() { ... }
    public Security getSecurity() { ... }

    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() { ... }
        public String getPassword() { ... }
        public List<String> getRoles() { ... }
    }
}
In this setup, the @ConstructorBinding annotation is used to indicate that constructor binding should be used. This means that the binder will expect to find a constructor with the parameters that you wish to have bound.

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

Default values can be specified using @DefaultValue and the same conversion service will be applied to coerce the String value to the target type of a missing property. By default, if no properties are bound to Security, the AcmeProperties instance will contain a null value for security. If you wish you return a non-null instance of Security even when no properties are bound to it, you can use an empty @DefaultValue annotation to do so:

```java
package com.example;
import java.net.InetAddress;
import java.util.List;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.ConstructorBinding;
import org.springframework.boot.context.properties.bind.DefaultValue;
@ConstructorBinding
@ConfigurationProperties("acme")
public class AcmeProperties {
    private final boolean enabled;
    private final InetAddress remoteAddress;
    private final Security security;
    public AcmeProperties(boolean enabled, InetAddress remoteAddress, @DefaultValue Security security) {
        this.enabled = enabled;
        this.remoteAddress = remoteAddress;
        this.security = security;
    }
}
```

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 (e.g. @Component beans, beans created via @Bean methods or beans loaded using @Import)
If you have more than one constructor for your class you can also use @ConstructorBinding directly on the constructor that should be bound.

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
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(AcmeProperties.class)
public class MyConfiguration {
}
```

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

When the @ConfigurationProperties bean is registered using configuration property scanning or via @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.

The bean name in the example above is acme-com.example.AcmeProperties.
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:

```yml
acme:
  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
@Service
public class MyService {

  private final AcmeProperties properties;

  @Autowired
  public MyService(AcmeProperties properties) {
    this.properties = properties;
  }

  //...

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

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
@ConfigurationProperties(prefix = "another")
@Bean
public AnotherComponent anotherComponent() {
    ...
}
```

Any JavaBean property defined with the `another` prefix is mapped onto that `AnotherComponent` bean in manner similar to the preceding `AcmeProperties` 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
@ConfigurationProperties(prefix="acme.my-project.person")
public class OwnerProperties {
    private String firstName;
    
    public String getFirstName() {
        return this.firstName;
    }
    
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
}
```

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

*Table 6. relaxed binding*
## Property Source

<table>
<thead>
<tr>
<th>Property Source</th>
<th>Simple</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties Files</td>
<td>Camel case, kebab case, or underscore notation</td>
<td>Standard list syntax using [ ] or comma-separated values</td>
</tr>
<tr>
<td>YAML Files</td>
<td>Camel case, kebab case, or underscore notation</td>
<td>Standard YAML list syntax or comma-separated values</td>
</tr>
<tr>
<td>Environment Variables</td>
<td>Upper case format with underscore as the delimiter (see Binding from Environment Variables)</td>
<td>Numeric values surrounded by underscores (see Binding from Environment Variables)</td>
</tr>
<tr>
<td>System properties</td>
<td>Camel case, kebab case, or underscore notation</td>
<td>Standard list syntax using [ ] or comma-separated values</td>
</tr>
</tbody>
</table>

We recommend that, when possible, properties are stored in lower-case kebab format, such as `my.property-name=acme`.

### Binding Maps

When binding to Map properties, if the key contains anything other than lowercase alpha-numeric characters or -, you need to use the bracket notation so that the original 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:

#### Properties

```properties
acme.map.[/key1]=value1
acme.map.[/key2]=value2
acme.map./key3=value3
```
Yaml

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

The properties above will bind to a `Map` with `/key1`, `/key2` and `key3` as the keys in the map.

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

**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.acme[0].other` would use an environment variable named `MY_ACME_0_OTHER`.

**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 `AcmeProperties`:
@ConfigurationProperties("acme")
public class AcmeProperties {

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

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

Consider the following configuration:

Properties

```
acme.list[0].name=my name
acme.list[0].description=my description
#---
spring.config.activate.on-profile=dev
acme.list[0].name=my another name
```

Yaml

```
acme:
    list:
    - name: "my name"
        description: "my description"
    ---
    spring:
        config:
            activate:
                on-profile: "dev"
    acme:
    list:
    - name: "my another name"
```

If the dev profile is not active, AcmeProperties.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:
In the preceding example, if the `dev` profile is active, `AcmeProperties.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 `AcmeProperties`:

```java
@ConfigurationProperties("acme")
public class AcmeProperties {

    private final Map<String, MyPojo> map = new HashMap<>();

    public Map<String, MyPojo> getMap() {
        return this.map;
    }
}
```

Consider the following configuration:
Properties

```
acme.map.key1.name=my name 1
acme.map.key1.description=my description 1
#---
spring.config.activate.on-profile=dev
acme.map.key1.name=dev name 1
acme.map.key2.name=dev name 2
acme.map.key2.description=dev description 2
```

Yaml

```
acme:
  map:
    key1:
      name: "my name 1"
      description: "my description 1"

---

spring:
  config:
    activate:
      on-profile: "dev"

acme:
  map:
    key1:
      name: "dev name 1"
    key2:
      name: "dev name 2"
      description: "dev description 2"
```

If the dev profile is not active, `AcmeProperties.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`).

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`).
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 (e.g. `10s` means 10 seconds)

Consider the following example:

```java
@ConfigurationProperties("app.system")
public class AppSystemProperties {

    @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;
    }
}
```

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
@ConfigurationProperties("app.system")
@ConstructorBinding
public class AppSystemProperties {

    private final Duration sessionTimeout;
    private final Duration readTimeout;

    public AppSystemProperties(@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;
    }
}
```

If you are upgrading a `Long` property, make sure to define the unit (using `@DurationUnit`) if it isn’t 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 (e.g. 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

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 (e.g. 10MB means 10 megabytes)

Consider the following example:
@ConfigurationProperties("app.io")
public class AppIoProperties {

    @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;
    }
}

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:
@ConfigurationProperties("app.io")
@ConstructorBinding
public class AppIoProperties {
    private final DataSize bufferSize;
    private final DataSize sizeThreshold;

    public AppIoProperties(@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;
    }
}

If you are upgrading a Long property, make sure to define the unit (using @DataSizeUnit) if it isn't 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 javax.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:

@ConfigurationProperties(prefix="acme")
@Validated
public class AcmeProperties {
    @NotNull
    private InetAddress remoteAddress;

    // ... getters and setters
}
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 `AcmeProperties` example:

```java
@ConfigurationProperties(prefix="acme")
@Validated
public class AcmeProperties {

    @NotNull
    private InetAddress remoteAddress;

    @Valid
    private final Security security = new Security();

    // ... getters and setters

    public static class Security {
        @NotEmpty
        public String username;

        // ... getters and setters
    }
}
```

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.

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>@ConfigurationProperties</th>
<th>@Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relaxed binding</td>
<td>Yes</td>
<td>Limited (see note below)</td>
</tr>
<tr>
<td>Meta-data support</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SpEL evaluation</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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}"), 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 a 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 via @Value.

### 4.3. 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
@Configuration(proxyBeanMethods = false)
@Profile("production")
public class ProductionConfiguration {

    // ...
}
```

If @ConfigurationProperties beans are registered via @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**

```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`.

### 4.3.1. 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 `SpringApplication` entry point has a Java API for setting additional profiles (that is, on top of those activated by the `spring.profiles.active` property). See the `setAdditionalProfiles()` method in `SpringApplication`. Profile groups, which are described in the next section can also be used to add active profiles if a given profile is active.

### 4.3.2. 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
```
Our application can now be started using `--spring.profiles.active=production` to activate the production, proddb and prodmq profiles in one hit.

### 4.3.3. Programatically Setting Profiles

You can programatically set active profiles by calling `SpringApplication.setAdditionalProfiles(…)` before your application runs. It is also possible to activate profiles by using Spring's `ConfigurableEnvironment` interface.

### 4.3.4. Profile-specific Configuration Files

Profile-specific variants of both `application.properties` (or `application.yml`) and files referenced through `@ConfigurationProperties` are considered as files and loaded. See "Profile Specific Files" for details.

### 4.4. 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.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When you deploy your application to a servlet container or application server, logging performed via 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.</td>
<td></td>
</tr>
</tbody>
</table>
4.4.1. Log Format

The default log output from Spring Boot resembles the following example:

```
2019-03-05 10:57:51.112  INFO 45469 --- [          main]
org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/7.0.52
2019-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2019-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1]
o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1358 ms
2019-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1]
o.s.b.c.e.ServletRegistrationBean        : Mapping servlet: 'dispatcherServlet' to 
2019-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1]
o.s.b.c.embedded.FilterRegistrationBean  : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
```

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.

Logback does not have a FATAL level. It is mapped to ERROR.

4.4.2. 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
```

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:

<table>
<thead>
<tr>
<th>Level</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>FATAL</td>
<td>Red</td>
</tr>
<tr>
<td>ERROR</td>
<td>Red</td>
</tr>
<tr>
<td>WARN</td>
<td>Yellow</td>
</tr>
<tr>
<td>INFO</td>
<td>Green</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Green</td>
</tr>
<tr>
<td>TRACE</td>
<td>Green</td>
</tr>
</tbody>
</table>

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 HH:mm:ss.SSS}){yellow}
```

The following colors and styles are supported:

- blue
- cyan
- faint
- green
- magenta
- red
- yellow

**4.4.3. 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>
<thead>
<tr>
<th>logging.file.name</th>
<th>logging.file.path</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>(none)</td>
<td></td>
<td>Console only logging.</td>
</tr>
<tr>
<td>Specific file</td>
<td>(none)</td>
<td>my.log</td>
<td>Writes to the specified log file. Names can be an exact location or relative to the current directory.</td>
</tr>
<tr>
<td>(none)</td>
<td>Specific directory</td>
<td>/var/log</td>
<td>Writes <code>spring.log</code> to the specified directory. Names can be an exact location or relative to the current directory.</td>
</tr>
</tbody>
</table>

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.

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.

### 4.4.4. File Rotation

If you are using the Logback, it’s possible to fine-tune log rotation settings using your `application.properties` or `application.yaml` file. For all other logging system, you’ll need to configure rotation settings directly yourself (for example, if you use Log4J2 then you could add a `log4j.xml` file).

The following rotation policy properties are supported:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>logging.logback.rollingpolicy.file-name-pattern</code></td>
<td>The filename pattern used to create log archives.</td>
</tr>
<tr>
<td><code>logging.logback.rollingpolicy.clean-history-on-start</code></td>
<td>If log archive cleanup should occur when the application starts.</td>
</tr>
<tr>
<td><code>logging.logback.rollingpolicy.max-file-size</code></td>
<td>The maximum size of log file before it's archived.</td>
</tr>
<tr>
<td><code>logging.logback.rollingpolicy.total-size-cap</code></td>
<td>The maximum amount of size log archives can take before being deleted.</td>
</tr>
<tr>
<td><code>logging.logback.rollingpolicy.max-history</code></td>
<td>The number of days to keep log archives (defaults to 7)</td>
</tr>
</tbody>
</table>

### 4.4.5. 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's also possible to set logging levels using environment variables. For example, LOGGING_LEVEL_ORG_SPRINGFRAMEWORK_WEB=DEBUG will set org.springframework.web to DEBUG.

The above approach will only work for package level logging. Since relaxed binding always converts environment variables to lowercase, it's 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.

4.4.6. Log Groups

It's 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't easily remember top level packages.

To help with this, Spring Boot allows you to define logging groups in your Spring Environment. For example, here's 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Loggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td>org.springframework.jdbc.core, org.hibernate.SQL, org.jooq.tools.LoggerListener</td>
</tr>
</tbody>
</table>

### 4.4.7. 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`.

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 via System properties.

Depending on your logging system, the following files are loaded:

<table>
<thead>
<tr>
<th>Logging System</th>
<th>Customization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logback</td>
<td>logback-spring.xml, logback-spring.groovy, logback.xml, or logback.groovy</td>
</tr>
<tr>
<td>Log4j2</td>
<td>log4j2-spring.xml or log4j2.xml</td>
</tr>
<tr>
<td>JDK (Java Util Logging)</td>
<td>logging.properties</td>
</tr>
</tbody>
</table>
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.

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, as described in the following table:

<table>
<thead>
<tr>
<th>Spring Environment</th>
<th>System Property</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>logging.exception-conversion-word</td>
<td>LOG_EXCEPTION_CONVERSION_WORD</td>
<td>The conversion word used when logging exceptions.</td>
</tr>
<tr>
<td>logging.file.name</td>
<td>LOG_FILE</td>
<td>If defined, it is used in the default log configuration.</td>
</tr>
<tr>
<td>logging.file.path</td>
<td>LOG_PATH</td>
<td>If defined, it is used in the default log configuration.</td>
</tr>
<tr>
<td>logging.pattern.console</td>
<td>CONSOLE_LOG_PATTERN</td>
<td>The log pattern to use on the console (stdout).</td>
</tr>
<tr>
<td>logging.pattern.dateformat</td>
<td>LOG_DATEFORMAT_PATTERN</td>
<td>Appender pattern for log date format.</td>
</tr>
<tr>
<td>logging.charset.console</td>
<td>CONSOLE_LOG_CHARSET</td>
<td>The charset to use for console logging.</td>
</tr>
<tr>
<td>logging.pattern.file</td>
<td>FILE_LOG_PATTERN</td>
<td>The log pattern to use in a file (if LOG_FILE is enabled).</td>
</tr>
<tr>
<td>logging.charset.file</td>
<td>FILE_LOG_CHARSET</td>
<td>The charset to use for file logging (if LOG_FILE is enabled).</td>
</tr>
<tr>
<td>logging.pattern.level</td>
<td>LOG_LEVEL_PATTERN</td>
<td>The format to use when rendering the log level (default %5p).</td>
</tr>
<tr>
<td>PID</td>
<td>PID</td>
<td>The current process ID (discovered if possible and when not already defined as an OS environment variable).</td>
</tr>
</tbody>
</table>

If you're using Logback, the following properties are also transferred:
### Spring Environment

<table>
<thead>
<tr>
<th>Spring Environment</th>
<th>System Property</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>logging.logback.rollingpolicy.file-name-pattern</td>
<td>LOGBACK_ROLLINGPOLICY_FILE_NAME_PATTERN</td>
<td>Pattern for rolled-over log file names (default <code>${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz</code>).</td>
</tr>
<tr>
<td>logging.logback.rollingpolicy.clean-history-on-start</td>
<td>LOGBACK_ROLLINGPOLICY_CLEAN_HISTORY_ON_START</td>
<td>Whether to clean the archive log files on startup.</td>
</tr>
<tr>
<td>logging.logback.rollingpolicy.max-file-size</td>
<td>LOGBACK_ROLLINGPOLICY_MAX_FILE_SIZE</td>
<td>Maximum log file size.</td>
</tr>
<tr>
<td>logging.logback.rollingpolicy.total-size-cap</td>
<td>LOGBACK_ROLLINGPOLICY_TOTAL_SIZE_CAP</td>
<td>Total size of log backups to be kept.</td>
</tr>
<tr>
<td>logging.logback.rollingpolicy.max-history</td>
<td>LOGBACK_ROLLINGPOLICY_MAX_HISTORY</td>
<td>Maximum number of archive log files to keep.</td>
</tr>
</tbody>
</table>

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

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 `:-`.

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:%{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
```

### 4.4.8. 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.

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.
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 reference guide for more details. The following listing shows three sample profiles:

```xml
<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:
4.5. 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.

The auto-configuration applies when the default properties file for the configured resource bundle is available (i.e. `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**

```properties
spring.messages.basename=messages,config.i18n.messages
spring.messages.fallback-to-system-locale=false
```

**Yaml**

```yaml
spring:
  messages:
    basename: "messages,config.i18n.messages"
    fallback-to-system-locale: false
```

`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.

4.6. JSON

Spring Boot provides integration with three JSON mapping libraries:
4.6.1. 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.

4.6.2. 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.

4.6.3. 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 Apache Johnzon for which dependency management is provided.

4.7. Developing Web Applications

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.

4.7.1. 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:
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @RequestMapping(value="/\{user\}\" method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/\{user\}/customers\"", method=RequestMethod.GET)
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/\{user\}\", method=RequestMethod.DELETE)
    public User deleteUser(@PathVariable Long user) {
        // ...
    }
}

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.

**Spring MVC Auto-configuration**

Spring Boot provides auto-configuration for Spring MVC that works well with most applications.

The auto-configuration adds the following features on top of Spring's defaults:

- 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.

If you want to take complete control of Spring MVC, you can add your own @Configuration annotated with @EnableWebMvc, or alternatively add your own @Configuration-annotated DelegatingWebMvcConfiguration as described in the Javadoc of @EnableWebMvc.

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.context.annotation.*;
import org.springframework.http.converter.*;

@Configuration(proxyBeanMethods = false)
public class MyConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = ...
        HttpMessageConverter<?> another = ...
        return new 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.

Custom JSON 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:
import java.io.*;
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import org.springframework.boot.jackson.*;

@JsonComponent
public class Example {

    public static class Serializer extends JsonSerializer<SomeObject> {
        // ...
    }

    public static class Deserializer extends JsonDeserializer<SomeObject> {
        // ...
    }
}

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.

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 also enabled and 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:
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. Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format.

---

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.

---

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-a2d595e6ed9a0b24f027f2b63b134d6.css"/>`, in URLs:

```
Properties

spring.web.resources.chain.strategy.content.enabled=true
spring.web.resources.chain.strategy.content.paths=/**
```
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 `ResourceUrlParser`.

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:

### Yaml

```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 ResourceProperties for more supported options.

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.

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" won't 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 don't 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:

```
spring:
  mvc:
    contentnegotiation:
      favor-parameter: true
      parameter-name: "mynparam"
```
Most standard media types are supported out-of-the-box, but you can also define new ones:

**Properties**

```java
spring.mvc.contentnegotiation.media-types.markdown=text/markdown
```

**Yaml**

```yaml
spring:
  mvc:
    contentnegotiation:
      media-types:
        markdown: "text/markdown"
```

Suffix pattern matching is deprecated and will be removed in a future release. If you understand the caveats and would still like your application to use suffix pattern matching, the following configuration is required:

```java
spring.mvc.contentnegotiation.favor-path-extension=true
spring.mvc.pathmatch.use-suffix-pattern=true
```

Alternatively, rather than open all suffix patterns, it's more secure to only support registered suffix patterns:

**Properties**

```java
spring.mvc.contentnegotiation.favor-path-extension=true
spring.mvc.pathmatch.use-registered-suffix-pattern=true
```

**Yaml**

```yaml
spring:
  mvc:
    contentnegotiation:
      favor-path-extension: true
    pathmatch:
      use-registered-suffix-pattern: true
```

As of Spring Framework 5.3, Spring MVC supports several implementation strategies for matching request paths to Controller handlers. It was previously only supporting the `AntPathMatcher` strategy, but it now also offers `PathPatternParser`. Spring Boot now provides a configuration property to choose and opt in the new strategy:

**Properties**

```java
spring.mvc.pathmatch.matching-strategy=path-pattern-parser
```
For more details on why you should consider this new implementation, please check out the dedicated blog post.

PathPatternParser is an optimized implementation but restricts usage of some path patterns variants and is incompatible with suffix pattern matching (`spring.mvc.pathmatch.use-suffix-pattern`, `spring.mvc.pathmatch.use-registered-suffix-pattern`) or mapping the DispatcherServlet with a Servlet prefix (`spring.mvc.servlet.path`).

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

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`.

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.

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.

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
@ControllerAdvice(basePackageClasses = AcmeController.class)
public class AcmeControllerAdvice extends ResponseEntityExceptionHandler {
    @ExceptionHandler(YourException.class)
    @ResponseBody
    ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) {
        HttpStatus status = getStatus(request);
        return new ResponseEntity<>(new CustomErrorType(status.value(),
            ex.getMessage()), status);
    }

    private HttpStatus getStatus(HttpServletRequest request) {
        Integer statusCode = (Integer)
            request.getAttribute("javax.servlet.error.status_code");
        if (statusCode == null) {
            return HttpStatus.INTERNAL_SERVER_ERROR;
        }
        return HttpStatus.valueOf(statusCode);
    }
}
```

In the preceding example, if YourException is thrown by a controller defined in the same package as AcmeController, a JSON representation of the CustomErrorType POJO is used instead of the
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
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
        return ...
    }
}
```
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
@Bean
public ErrorPageRegistrar errorPageRegistrar(){
    return new MyErrorPageRegistrar();
}

// ...

private static class MyErrorPageRegistrar implements ErrorPageRegistrar {

    @Override
    public void registerErrorPages(ErrorPageRegistry registry) {
        registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    }
}
```

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
@Bean
public FilterRegistrationBean myFilter() {
    FilterRegistrationBean registration = new FilterRegistrationBean();
    registration.setFilter(new MyFilter());
    ...
    registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.class));
    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`.

If you are using Spring Security and want to access the principal in an error page, you must configure Spring Security's filter to be invoked on error dispatches. To do so, set the `spring.security.filter.dispatcher-types` property to `async, error, forward, request`.

**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.

**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
@Configuration(proxyBeanMethods = false)
public class MyConfiguration {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**");
            }
        };
    }
}
```
4.7.2. 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
@RestController
@RequestMapping("/users")
public class MyRestController {

    @GetMapping("/{user}")
    public Mono<User> getUser(@PathVariable Long user) {
        // ...
    }

    @GetMapping("/{user}/customers")
    public Flux<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @DeleteMapping("/{user}")
    public Mono<User> deleteUser(@PathVariable Long user) {
        // ...
    }
}
```

“WebFlux.fn”, the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:
@Configuration(proxyBeanMethods = false)
public class RoutingConfiguration {

    @Bean
    public RouterFunction<ServerResponse> monoRouterFunction(UserHandler userHandler) {
        return route(GET("/{user}"),
                     userHandler::getUser)
               .andRoute(GET("/{user}/customers"),
                     userHandler::getUserCustomers)
               .andRoute(DELETE("/{user}"),
                     userHandler::deleteUser);  
    }
}

@Component
public class UserHandler {

    public Mono<ServerResponse> getUser(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> getUserCustomers(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> deleteUser(ServerRequest request) {
        // ...
    }
}

WebFlux is part of the Spring Framework and detailed information is available in its reference documentation.

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.

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`.

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;

@Configuration(proxyBeanMethods = false)
public class MyConfiguration {

    @Bean
    public CodecCustomizer myCodecCustomizer() {
        return codecConfigurer -> {
            // ...
        };
    }
}
```

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. Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the WebJars format.

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).

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 a `WebExceptionHandler` 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
public class CustomErrorWebExceptionHandler extends AbstractErrorWebExceptionHandler {
    // Define constructor here

    @Override
    protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes errorAttributes) {
        return RouterFunctions
                .route(aPredicate, aHandler)
                .andRoute(anotherPredicate, anotherHandler);
    }
}
```

For a more complete picture, you can also subclass `DefaultErrorWebExceptionHandler` directly and override specific methods.

### 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 built with 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:
To map all 5xx errors by using a Mustache template, your directory structure would be as follows:

```
src/
 +- main/
   +- java/
     | + <source code>
   +- resources/
     +- public/
     | +- error/
     |   +- 404.html
     | +- <other public assets>
   +- 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:

<table>
<thead>
<tr>
<th>Web Filter</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>MetricsWebFilter</td>
<td>Ordered.HIGHEST_PRECEDENCE + 1</td>
</tr>
<tr>
<td>WebFilterChainProxy (Spring Security)</td>
<td>-100</td>
</tr>
<tr>
<td>HttpTraceWebFilter</td>
<td>Ordered.LOWEST_PRECEDENCE - 10</td>
</tr>
</tbody>
</table>

#### 4.7.3. 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:

```java
@Component
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
        register(Endpoint.class);
    }
}
```

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:

```java
@Component
@Path("/hello")
public class Endpoint {
    @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. Both the servlet and the filter registrations can be given init
parameters by using `spring.jersey.init.*` to specify a map of properties.

### 4.7.4. Embedded Servlet Container Support

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`.

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.

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 Servlet 3.0+ `javax.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.
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`.

```
@ServletComponentScan
```

This annotation 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.

```
@ServletComponentScan
```

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.

**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
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.stereotype.Component;

@Component
public class CustomizationBean implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory server) {
        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
@Component
public class TomcatServerCustomizerExample implements WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    @Override
    public void customize(TomcatServletWebServerFactory server) {
        server.addConnectorCustomizers(
            (tomcatConnector) ->
                tomcatConnector.setAsyncTimeout(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.

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.

4.7.5. 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.

4.7.6. 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.
4.8. 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.

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

```properties
server.shutdown=graceful
```

Yaml

```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

```properties
spring.lifecycle.timeout-per-shutdown-phase=20s
```

Yaml

```yaml
spring:
  lifecycle:
    timeout-per-shutdown-phase: "20s"
```

Using graceful shutdown with your IDE may not work properly if it does not send a proper SIGTERM signal. Refer to the documentation of your IDE for more details.

4.9. RSocket

RSocket is a binary protocol for use on byte stream transports. It enables symmetric interaction models via 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.

### 4.9.1. 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. Check out 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.

### 4.9.2. 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 (i.e. of type `WebApplicationType.REACTIVE`), the RSocket server will be plugged into the Web Server only if the following properties match:

#### Properties

```properties
spring.rsocket.server.mapping-path=/rsocket
spring.rsocket.server.transport=websocket
```

#### Yaml

```yaml
spring:
  rsocket:
    server:
      mapping-path: "/rsocket"
      transport: "websocket"
```

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**

```yaml
spring.rsocket.server.port=9898
```

**Yaml**

```yaml
spring:
  rsocket:
    server:
      port: 9898
```

### 4.9.3. 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.

### 4.9.4. 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.

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 shouldn’t create requesters with different setups using the same instance.

The following code shows a typical example:
@Service
public class MyService {
    private final Mono<RSocketRequester> rsocketRequester;

    public MyService(RSocketRequester.Builder rsocketRequesterBuilder) {
        this.rsocketRequester = rsocketRequesterBuilder.
                connectTcp("example.org", 9898).cache();
    }

    public Mono<User> someRSocketCall(String name) {
        return this.rsocketRequester.flatMap(req ->
                req.route("user").data(name).retrieveMono(User.class));
    }
}

4.10. 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 INFO level when the application starts, as shown in the following example:

Using generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35

If you fine-tune your logging configuration, ensure that the org.springframework.boot.autoconfigure.security category is set to log INFO-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.

### 4.10.1. 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` or `WebSecurityConfigurerAdapter` 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.

### 4.10.2. 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:
@Bean
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    return http
        .authorizeExchange()
            .matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
            .pathMatchers("/foo", "/bar")
            .authenticated().and()
            .formLogin().and()
            .build();
}

4.10.3. 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 an 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-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=https://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-1.client-authentication-method=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.client-authentication-method=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/oauth/authorize
spring.security.oauth2.client.provider.my-oauth-provider.token-uri=https://my-auth-server/oauth/token
spring.security.oauth2.client.provider.my-oauth-provider.user-info-uri=https://my-auth-server/userinfo
spring.security.oauth2.client.provider.my-oauth-provider.user-info-authentication-method=header
spring.security.oauth2.client.provider.my-oauth-provider.user-name-attribute=name
Yaml

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          my-client-1:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for user scope"
            provider: "my-oauth-provider"
            scope: "user"
            redirect-uri: "https://my-redirect-uri.com"
            client-authentication-method: "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: "https://my-redirect-uri.com"
            client-authentication-method: "basic"
            authorization-grant-type: "authorization_code"
        provider:
          my-oauth-provider:
            authorization-uri: "https://my-auth-server/oauth/authorize"
            token-uri: "https://my-auth-server/oauth/token"
            user-info-uri: "https://my-auth-server/userinfo"
            user-info-authentication-method: "header"
            jwk-set-uri: "https://my-auth-server/token_keys"
            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 the provider 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

```
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
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .anyRequest().authenticated()
        .and()
        .oauth2Login()
        .redirectionEndpoint()
        .baseUri("/custom-callback");
    return http.build();
}
```

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/
```
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 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 via introspection:

**Properties**

```yaml
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

```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**

Currently, Spring Security does not provide support for implementing an OAuth 2.0 Authorization Server.
Server. However, this functionality is available from the Spring Security OAuth project, which will eventually be superseded by Spring Security completely. Until then, you can use the spring-security-oauth2-autoconfigure module to easily set up an OAuth 2.0 authorization server; see its documentation for instructions.

4.10.4. 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.credential...private-key
spring.security.saml2.relyingparty.registration.my-relying-party1.signing.credential...certificate
spring.security.saml2.relyingparty.registration.my-relying-party1.decryption.credential...private-key
spring.security.saml2.relyingparty.registration.my-relying-party1.decryption.credential...certificate
spring.security.saml2.relyingparty.registration.my-relying-party1.identityprovider.verification.credential...certificate
spring.security.saml2.relyingparty.registration.my-relying-party1.identityprovider.entity-id=remote-idp-entity-id1
spring.security.saml2.relyingparty.registration.my-relying-party1.identityprovider.sso-url=https://remoteidp1.sso.url
spring.security.saml2.relyingparty.registration.my-relying-party2.signing.credential...private-key
spring.security.saml2.relyingparty.registration.my-relying-party2.signing.credential...certificate
spring.security.saml2.relyingparty.registration.my-relying-party2.decryption.credential...private-key
spring.security.saml2.relyingparty.registration.my-relying-party2.decryption.credential...certificate
spring.security.saml2.relyingparty.registration.my-relying-party2.identityprovider.verification.credential...certificate
spring.security.saml2.relyingparty.registration.my-relying-party2.identityprovider.entity-id=remote-idp-entity-id2
spring.security.saml2.relyingparty.registration.my-relying-party2.identityprovider.sso-url=https://remoteidp2.sso.url
```
Yaml

```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"
          identityprovider:
            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"
          identityprovider:
            verification:
              credentials:
                - certificate-location: "path-to-other-verification-cert"
            entity-id: "remote-idp-entity-id2"
            sso-url: "https://remoteidp2.sso.url"
```

4.10.5. Actuator Security

For security purposes, all actuators other than /health and /info are disabled by default. The `management.endpoints.web.exposure.include` property can be used to enable the actuators.

If Spring Security is on the classpath and no other `WebSecurityConfigurerAdapter` or `SecurityFilterChain` bean is present, all actuators other than /health and /info are secured by Spring Boot auto-configuration. If you define a custom `WebSecurityConfigurerAdapter` or `SecurityFilterChain` bean, Spring Boot auto-configuration will back off and you will be in full control of actuator access rules.
Before setting the `management.endpoints.web.exposure.include`, ensure that the exposed actuators do not contain sensitive information and/or are secured by placing them behind a firewall or by something like Spring Security.

**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), `PUT` or `DELETE` will get a 403 forbidden error when the default security configuration is in use.

We recommend disabling CSRF protection completely only if you are creating a service that is used by non-browser clients.

Additional information about CSRF protection can be found in the [Spring Security Reference Guide](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/).

### 4.11. Working with SQL Databases

The [Spring Framework](https://spring.io/blog/2011/10/19/spring-3-0-available-now) 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](https://projects.spring.io/spring-data/) provides an additional level of functionality: creating `Repository` implementations directly from interfaces and using conventions to generate queries from your method names.

#### 4.11.1. 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.

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.

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 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:

```xml
<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>
```

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`.

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`. 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.

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. If you define your own `DataSource` bean, auto-configuration does not occur.

DataSource configuration is controlled by external configuration properties in `spring.datasource.*`. For example, you might declare the following section in `application.properties`:

**Properties**

```properties
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
```

**Yaml**

```yaml
spring:
  datasource:
    url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    password: "dbpass"
```

You should at least specify the URL by setting the `spring.datasource.url` property. Otherwise, Spring Boot tries to auto-configure an embedded database.

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.

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.*`). Refer to 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**

```properties
spring.datasource.tomcat.max-wait=10000
spring.datasource.tomcat.max-active=50
spring.datasource.tomcat.test-on-borrow=true
```
This will set the pool to wait 10000 ms 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.

**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"
```

### 4.11.2. 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:
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public MyBean(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    // ...
}

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
```

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.

4.11.3. 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.
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, all packages below your main configuration class (the one annotated with @EnableAutoConfiguration or @SpringBootApplication) are searched.

Any classes annotated with @Entity, @Embeddable, or @MappedSuperclass are considered. A typical entity class resembles the following example:
package com.example.myapp.domain;

import java.io.Serializable;
import javax.persistence.*

@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 shouldn't 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
}

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, repositories are searched from the package containing your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) down.

The following example shows a typical Spring Data repository interface definition:

```java
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {
    Page<City> findAll(Pageable pageable);
    City findByNameAndStateAllIgnoringCase(String name, String state);
}
```

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.

When using deferred or lazy bootstrapping, make sure to defer any access to the JPA infrastructure after the application context bootstrap phase.

We have barely scratched the surface of Spring Data JPA. For complete details, see the Spring Data JPA 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
```
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:

```
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`.

### 4.11.4. 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 a `JdbcConfiguration` subclass to your application.

For complete details of Spring Data JDBC, please refer to the [reference documentation](#).

### 4.11.5. 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](#).
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`.

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.

### 4.11.6. 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:
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 `@Autowire` it, as shown in the following example:

```java
@Component
public class JooqExample implements CommandLineRunner {
    private final DSLContext create;

    @Autowired
    public JooqExample(DSLContext dslContext) {
        this.create = dslContext;
    }
}
```

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:
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);
}

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.

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 @Bean definitions, which is used when the jOOQ Configuration is created. You can define beans for the following jOOQ Types:

- ConnectionProvider
- ExecutorProvider
- TransactionProvider
- RecordMapperProvider
- RecordUnmapperProvider
- Settings
- RecordListenerProvider
- ExecuteListenerProvider
- VisitListenerProvider
- TransactionListenerProvider

You can also create your own org.jooq.Configuration @Bean if you want to take complete control of the jOOQ configuration.

4.11.7. 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 via 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"

---

You do not need to specify a driver class name, since Spring Boot obtains the driver from R2DBC's Connection Factory discovery.

At least the url should be provided. Information specified in the URL takes precedence over individual properties, i.e. name, username, password and pooling options.

The “How-to” section includes a section on how to initialize a database.

To customize the connections created by a ConnectionFactory, i.e., 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 is taken from the application configuration:

```java
@Bean
class ConnectionFactoryPortCustomizer() {
  return (builder) -> builder.option(PORT, 5432);
}
```

The following examples show how to set some PostgreSQL connection options:

```java
@Bean
class PostgresCustomizer() {
  Map<String, String> options = new HashMap<>();
  options.put("lock_timeout", "30s");
  options.put("statement_timeout", "60s");
  return (builder) -> builder.option(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:

```xml
<dependency>
  <groupId>io.r2dbc</groupId>
  <artifactId>r2dbc-h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

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 `@Autowired` it directly into your own beans, as shown in the following example:

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.r2dbc.function.DatabaseClient;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final DatabaseClient databaseClient;

    @Autowired
    public MyBean(DatabaseClient databaseClient) {
        this.databaseClient = databaseClient;
    }

    // ...
}
```

**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, repositories are searched from the package containing your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) down.

The following example shows a typical Spring Data repository interface definition:

```java
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;
import reactor.core.publisher.Mono;

public interface CityRepository extends Repository<City, Long> {
    Mono<City> findByNameAndStateAllIgnoringCase(String name, String state);
}
```

We have barely scratched the surface of Spring Data R2DBC. For complete details, see the [Spring Data R2DBC reference documentation](https://docs.spring.io/spring-data/r2dbc/docs).

### 4.12. Working with NoSQL Technologies

Spring Data provides additional projects that help you access a variety of NoSQL technologies, including:

- **MongoDB**
- **Neo4j**
- **Elasticsearch**
- **Solr**
- **Redis**
- **GemFire** or **Geode**
- **Cassandra**
- **Couchbase**
- **LDAP**

Spring Boot provides auto-configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr, Cassandra, Couchbase, and LDAP. You can make use of the other projects, but you must configure them yourself. Refer to the appropriate reference documentation at [spring.io/projects/spring-data](https://spring.io/projects/spring-data).
4.12.1. 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.

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. By default, the instance tries to connect to a Redis server at localhost:6379. The following listing shows an example of such a bean:

```java
@Component
public class MyBean {

    private StringRedisTemplate template;

    @Autowired
    public MyBean(StringRedisTemplate template) {
        this.template = template;
    }

    // ...
}
```

You can also register an arbitrary number of beans that implement LettuceClientConfigurationBuilderCustomizer for more advanced customizations. If you use Jedis, JedisClientConfigurationBuilderCustomizer is also available.

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, if commons-pool2 is on the classpath, you get a pooled connection factory.

4.12.2. 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 org.springframework.data.mongodb.MongoDatabaseFactory;
import com.mongodb.client.MongoDatabase;

@Component
public class MyBean {

    private final MongoDatabaseFactory mongo;

    @Autowired
    public MyBean(MongoDatabaseFactory mongo) {
        this.mongo = mongo;
    }

    // ...

    public void example() {
        MongoDatabase db = mongo.getMongoDatabase();
        // ...
    }
}
```

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:

```
spring.data.mongodb.uri=mongodb://user:secret@mongo1.example.com:12345,mongo2.example.com:23456/test
```

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=mongoserver.example.com
spring.data.mongodb.port=27017
spring.data.mongodb.database=test
spring.data.mongodb.username=user
spring.data.mongodb.password=secret

Yaml

```yaml
spring:
  data:
    mongodb:
      host: "mongoserver.example.com"
      port: 27017
      database: "test"
      username: "user"
      password: "secret"
```

- If `spring.data.mongodb.port` is not specified, the default of `27017` is used. You could delete this line from the example shown earlier.

- 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.

- 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 hasn’t 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:
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;
    }

    // ...
}

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
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndStateAllIgnoringCase(String name, String state);
}
```

You can customize document scanning locations by using the @EntityScan annotation.

For complete details of Spring Data MongoDB, including its rich object mapping technologies, refer to its reference documentation.
Embedded Mongo

Spring Boot offers auto-configuration for Embedded Mongo. To use it in your Spring Boot application, add a dependency on `de.flapdoodle.embed:de.flapdoodle.embed.mongo`.

The port that Mongo listens on can be configured by setting the `spring.data.mongodb.port` property. To use a randomly allocated free port, use a value of 0. The `MongoClient` created by `MongoAutoConfiguration` is automatically configured to use the randomly allocated port.

If you do not configure a custom port, the embedded support uses a random port (rather than 27017) by default.

If you have SLF4J on the classpath, the output produced by Mongo is automatically routed to a logger named `org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo`.

You can declare your own `IMongodConfig` and `IRuntimeConfig` beans to take control of the Mongo instance's configuration and logging routing. The download configuration can be customized by declaring a `DownloadConfigBuilderFactoryCustomizer` bean.

4.12.3. 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
@Component
public class MyBean {

    private final Driver driver;

    @Autowired
    public MyBean(Driver driver) {
        this.driver = driver;
    }

    // ...
}
```

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:
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, refer to 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
package com.example.myapp.domain;

import java.util.Optional;

import org.springframework.data.neo4j.repository.*;

public interface CityRepository extends Neo4jRepository<City, Long> {
    Optional<City> findOneByNameAndState(String name, String state);
}
```

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.

You can customize the locations to look for repositories and entities by using @EnableNeo4jRepositories and @EntityScan respectively on a @Configuration-bean.
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
@Bean
public ReactiveNeo4jTransactionManager reactiveTransactionManager(Driver driver,
    ReactiveDatabaseSelectionProvider databaseNameProvider) {
    return new ReactiveNeo4jTransactionManager(driver,
        databaseNameProvider);
}
```

### 4.12.4. Solr

Apache Solr is a search engine. Spring Boot offers basic auto-configuration for the Solr 5 client library and the abstractions on top of it provided by Spring Data Solr. There is a `spring-boot-starter-data-solr` "Starter" for collecting the dependencies in a convenient way.

#### Connecting to Solr

You can inject an auto-configured `SolrClient` instance as you would any other Spring bean. By default, the instance tries to connect to a server at `localhost:8983/solr`. The following example shows how to inject a Solr bean:

```java
@Component
public class MyBean {

    private SolrClient solr;

    @Autowired
    public MyBean(SolrClient solr) {
        this.solr = solr;
    }

    // ...
}
```

If you add your own `@Bean` of type `SolrClient`, it replaces the default.

#### Spring Data Solr Repositories

Spring Data includes repository support for Apache Solr. As with the JPA repositories discussed earlier, the basic principle is that queries are automatically constructed for you based on method names.

In fact, both Spring Data JPA and Spring Data Solr share the same common infrastructure. You
could take the JPA example from earlier and, assuming that `City` is now a `@SolrDocument` class rather than a JPA `@Entity`, it works in the same way.

IP: For complete details of Spring Data Solr, refer to the reference documentation.

### 4.12.5. Elasticsearch

Elasticsearch is an open source, distributed, RESTful search and analytics engine. Spring Boot offers basic auto-configuration for Elasticsearch.

Spring Boot supports several clients:

- The official Java "Low Level" and "High Level" REST clients
- 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 and the "High Level" client. Spring Boot provides support for the "High Level" client, which ships with `org.elasticsearch.client:elasticsearch-rest-high-level-client`.

If you have this dependency on the classpath, Spring Boot will auto-configure and register a `RestHighLevelClient` bean that by default targets `localhost:9200`. You can further tune how `RestHighLevelClient` is configured, as shown in the following example:

**Properties**

```properties
spring.elasticsearch.rest.uris=https://search.example.com:9200
spring.elasticsearch.rest.read-timeout=10s
spring.elasticsearch.rest.username=user
spring.elasticsearch.rest.password=secret
```

**Yaml**

```yaml
spring:
  elasticsearch:
    rest:
      uris: "https://search.example.com:9200"
      read-timeout: "10s"
      username: "user"
      password: "secret"
```

You can also register an arbitrary number of beans that implement `RestClientBuilderCustomizer` for more advanced customizations. To take full control over the registration, define a `RestClientBuilder` bean.
If your application needs access to a "Low Level" RestClient, you can get it by calling `client.getLowLevelClient()` on the auto-configured RestHighLevelClient.

**Connecting to Elasticsearch using Reactive REST clients**

Spring Data Elasticsearch ships ReactiveElasticsearchClient for querying Elasticsearch instances in a reactive fashion. It is built on top of WebFlux's WebClient, so both spring-boot-starter-elasticsearch and spring-boot-starter-webflux dependencies are useful to enable this support.

By default, Spring Boot will auto-configure and register a ReactiveElasticsearchClient bean that targets `localhost:9200`. You can further tune how it is configured, as shown in the following example:

**Properties**

```properties
spring.data.elasticsearch.client.reactive.endpoints=search.example.com:9200
spring.data.elasticsearch.client.reactive.use-ssl=true
spring.data.elasticsearch.client.reactive.socket-timeout=10s
spring.data.elasticsearch.client.reactive.username=user
spring.data.elasticsearch.client.reactive.password=secret
```

**Yaml**

```yaml
spring:
  data:
    elasticsearch:
      client:
        reactive:
          endpoints: "search.example.com:9200"
          use-ssl: true
          socket-timeout: "10s"
          username: "user"
          password: "secret"
```

If the configuration properties are not enough and you'd like to fully control the client configuration, you can register a custom ClientConfiguration bean.

**Connecting to Elasticsearch by Using Spring Data**

To connect to Elasticsearch, a RestHighLevelClient bean must be defined, auto-configured by Spring Boot or manually provided by the application (see previous sections). With this configuration in place, an ElasticsearchRestTemplate can be injected like any other Spring bean, as shown in the following example:
@Component
public class MyBean {
  private final ElasticsearchRestTemplate template;

  public MyBean(ElasticsearchRestTemplate template) {
    this.template = template;
  }

  // ...
}

In the presence of spring-data-elasticsearch and the required dependencies for using a WebClient
(typically spring-boot-starter-webflux), 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.

💡 For complete details of Spring Data Elasticsearch, refer to 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
4.12.6. 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.data.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.data.cassandra.keyspace-name=mykeyspace
spring.data.cassandra.contact-points=cassandrahost1:9042,cassandrahost2:9042
spring.data.cassandra.local-datacenter=datacenter1
```

Yaml

```
spring:
data:
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.data.cassandra.keyspace-name=mykeyspace
spring.data.cassandra.contact-points=cassandrahost1,cassandrahost2
spring.data.cassandra.local-datacenter=datacenter1
```
Those two examples are identical as the port default to 9042. If you need to configure the port, use `spring.data.cassandra.port`.

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 and rather provides a number of configuration properties via the `spring.data.cassandra.*` namespace. 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`.

If you're using `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
@Component
public class MyBean {

    private final CassandraTemplate template;

    public MyBean(CassandraTemplate template) {
        this.template = template;
    }

    // ...
}
```

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 to annotate finder methods with `@Query`. 
4.12.7. 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 use to open a new Bucket and enables SSL support:

Properties

```
spring.couchbase.env.timeouts.connect=3s
spring.couchbase.env.ssl.key-store=/location/of/keystore.jks
spring.couchbase.env.ssl.key-store-password=secret
```
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. For complete details of Spring Data Couchbase, refer to 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:
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
@Configuration(proxyBeanMethods = false)
public class SomeConfiguration {
    @Bean(BeaNames.COUCHBASE_CUSTOM_CONVERSIONS)
    public CustomConversions myCustomConversions() {
        return new CustomConversions(...);
    }

    // ...
}
```

### 4.12.8. 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

```ini
spring.ldap.urls=ldap://myserver:1235
spring.ldap.username=admin
spring.ldap.password=secret
```

Yaml

```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. For complete details of Spring Data LDAP, refer to 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:
@Component
public class MyBean {

    private final LdapTemplate template;

    @Autowired
    public MyBean(LdapTemplate template) {
        this.template = template;
    }

    // ...
}

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

```properties
spring.ldap.embedded.base-dn=dc=spring,dc=io
```

Yaml

```yaml
spring:
ldap:
    embedded:
        base-dn: "dc=spring,dc=io"
```

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

```properties
spring.ldap.embedded.base-dn[0]=dc=spring,dc=io
spring.ldap.embedded.base-dn[1]=dc=pivotal,dc=io
```

Yaml

```yaml
spring.ldap.embedded.base-dn:
    - dc=spring,dc=io
    - dc=pivotal,dc=io
```
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.

4.12.9. 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.

4.13. 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 via the `@EnableCaching` annotation.

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 MathService {
    @Cacheable("piDecimals")
    public int computePiDecimal(int i) {
        // ...
    }
}
```

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.

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.

It is also possible to transparently update or evict data from the cache.

### 4.13.1. 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. EhCache 2.x
4. Hazelcast
5. Infinispan
6. Couchbase
7. Redis
8. Caffeine
9. Simple

It is also possible to force a particular cache provider by setting the `spring.cache.type` property. Use this property if you need to disable caching altogether in certain environment (such as tests).

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, EhCache 2.x, 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 be passed down to the underlying map:

```java
@Bean
public CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer() {
    return new CacheManagerCustomizer<ConcurrentMapCacheManager>() {
        @Override
        public void customize(ConcurrentMapCacheManager cacheManager) {
            cacheManager.setAllowNullValues(false);
        }
    };
}
```

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**

```yaml
# Only necessary if more than one provider is present
spring.cache.jcache.provider=com.acme.MyCachingProvider
spring.cache.jcache.config=classpath:acme.xml
```

**Yaml**

```yaml
# Only necessary if more than one provider is present
spring:
  cache:
    jcache:
      provider: "com.acme.MyCachingProvider"
      config: "classpath:acme.xml"
```

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.

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.

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.
EhCache 2.x

EhCache 2.x is used if a file named `ehcache.xml` can be found at the root of the classpath. If EhCache 2.x is found, the `EhCacheCacheManager` provided by the `spring-boot-starter-cache` “Starter” is used to bootstrap the cache manager. An alternate configuration file can be provided as well, as shown in the following example:

**Properties**

```
spring.cache.ehcache.config=classpath:config/another-config.xml
```

**Yaml**

```
spring:
cache:
ehcache:
  config: "classpath:config/another-config.xml"
```

Hazelcast

Spring Boot has **general support for Hazelcast**. If a `HazelcastInstance` has been auto-configured, it is automatically wrapped in a `CacheManager`.

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.

ℹ️ The support of Infinispan in Spring Boot is restricted to the embedded mode and is quite basic. If you want more options, you should use the official Infinispan Spring Boot starter instead. See Infinispan’s documentation for more details.
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
@Bean
public CouchbaseCacheManagerBuilderCustomizer myCouchbaseCacheManagerBuilderCustomizer() {
  return (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:

---

173
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.

You can take full control of the default configuration by adding a RedisCacheConfiguration @Bean of your own. This can be useful if you’re looking for customizing 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
@Bean
public RedisCacheManagerBuilderCustomizer myRedisCacheManagerBuilderCustomizer() {
    return (builder) -> builder
        // configurations for cache1
        .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**

```properties
spring.cache.cache-names=cache1,cache2
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s
```

**Yaml**

```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.

**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**

```properties
spring.cache.cache-names=cache1,cache2
```

**Yaml**

```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 need to disable caching altogether in certain environments, force the cache type to `none` to use a no-op implementation, as shown in the following example:

**Properties**

```
spring.cache.type=none
```

**Yaml**

```
spring:
cache:
  type: "none"
```

### 4.14. 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.

#### 4.14.1. JMS

The `javax.jms.ConnectionFactory` interface provides a standard method of creating a `javax.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.

**ActiveMQ Support**

When `ActiveMQ` is available on the classpath, Spring Boot can also configure a `ConnectionFactory`. If the broker is present, an embedded broker is automatically started and configured (provided no broker URL is specified through configuration).

> If you use `spring-boot-starter-activemq`, the necessary dependencies to connect or embed an ActiveMQ instance are provided, as is the Spring infrastructure to integrate with JMS.

ActiveMQ configuration is controlled by external configuration properties in `spring.activemq.*`. For example, you might declare the following section in `application.properties`:
By default, a CachingConnectionFactory wraps the native ConnectionFactory with sensible settings that you can control by external configuration properties in spring.jms.*:

Properties

```properties
spring.jms.cache.session-cache-size=5
```

Yaml

```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

```properties
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=50
```

Yaml

```yaml
spring:
  activemq:
    pool:
      enabled: true
      max-connections: 50
```
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 creates a destination if it does not yet exist so that destinations are resolved against their provided names.

**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.

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-jms-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

```properties
spring.artemis.mode=native
spring.artemis.host=192.168.1.210
spring.artemis.port=9876
spring.artemis.user=admin
spring.artemis.password=secret
```

### Yaml

```yaml
spring:
  artemis:
    mode: native
    host: "192.168.1.210"
    port: 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 to 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 Artemis configuration or the names provided through configuration.

**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
```
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.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JmsTemplate jmsTemplate;

    @Autowired
    public MyBean(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

    // ...
}
```

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.

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 or a MessageConverter beans is defined, it is associated automatically to 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.

Yaml

```yaml
spring:
  jms:
    jndi-name: "java:/MyConnectionFactory"
```
The following component creates a listener endpoint on the `someQueue` destination:

```java
@Component
public class MyBean {
    @JmsListener(destination = "someQueue")
    public void processMessage(String content) {
        // ...
    }
}
```

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
@Configuration(proxyBeanMethods = false)
static class JmsConfiguration {
    @Bean
    public DefaultJmsListenerContainerFactory myFactory(
        DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory =
            new DefaultJmsListenerContainerFactory();
        configurer.configure(factory, connectionFactory());
        factory.setMessageConverter(myMessageConverter());
        return factory;
    }
}
```

Then you can use the factory in any `@JmsListener`-annotated method as follows:

```java
@Component
public class MyBean {
    @JmsListener(destination = "someQueue", containerFactory="myFactory")
    public void processMessage(String content) {
        // ...
    }
}
```
4.14.2. 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”.

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"
```

When specifying addresses that way, the host and port properties are ignored. If the address uses the amqps protocol, SSL support is enabled automatically.

If a ConnectionNameStrategy bean exists in the context, it will be automatically used to name connections created by the auto-configured ConnectionFactory. See RabbitProperties for more of the...
supported options.

See Understanding AMQP, the protocol used by RabbitMQ for more details.

### 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.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final AmqpAdmin amqpAdmin;
    private final AmqpTemplate amqpTemplate;

    @Autowired
    public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
        this.amqpAdmin = amqpAdmin;
        this.amqpTemplate = amqpTemplate;
    }

    // ...
}
```

`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
```
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.

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
@Component
public class MyBean {

    @RabbitListener(queues = "someQueue")
    public void processMessage(String content) {
        // ...
    }
}
```

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.

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
@Configuration(proxyBeanMethods = false)
static class RabbitConfiguration {

    @Bean
    public SimpleRabbitListenerContainerFactory myFactory(
        SimpleRabbitListenerContainerFactoryConfigurer configurer) {
        SimpleRabbitListenerContainerFactory factory =
            new SimpleRabbitListenerContainerFactory();
        configurer.configure(factory, connectionFactory);
        factory.setMessageConverter(myMessageConverter());
        return factory;
    }
}
```

Then you can use the factory in any `@RabbitListener`-annotated method, as follows:

```java
@Component
public class MyBean {

    @RabbitListener(queues = "someQueue", containerFactory="myFactory")
    public void processMessage(String content) {
        // ...
    }
}
```

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.

⚠️ 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.

### 4.14.3. 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**

```properties
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=myGroup
```

**Yaml**

```yaml
spring:
  kafka:
    bootstrap-servers: "localhost:9092"
    consumer:
      group-id: "myGroup"
```

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.

**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
@Component
public class MyBean {

  private final KafkaTemplate kafkaTemplate;

  @Autowired
  public MyBean(KafkaTemplate kafkaTemplate) {
    this.kafkaTemplate = kafkaTemplate;
  }

  // ...
}
```

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`.

**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
@Component
public class MyBean {
    @KafkaListener(topics = "someTopic")
    public void processMessage(String content) {
        // ...
    }
}
```

If a KafkaTransactionManager bean is defined, it is automatically associated to the container factory. Similarly, if a RecordFilterStrategy, ErrorHandler, 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.

A custom ChainedKafkaTransactionManager must be marked @Primary as it usually references the auto-configured KafkaTransactionManager bean.

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 via 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:
@Configuration(proxyBeanMethods = false)
@EnableKafkaStreams
public static class KafkaStreamsExampleConfiguration {

    @Bean
    public KStream<Integer, String> kStream(StreamsBuilder streamsBuilder) {
        KStream<Integer, String> stream = streamsBuilder.stream("ks1In");
        stream.map((k, v) -> new KeyValue<>(k, v.toUpperCase())).to("ks1Out",
            Produced.with(Serdes.Integer(), new JsonSerde<>());
        return stream;
    }
}

By default, the streams managed by the StreamBuilder object it creates are started automatically. You can customize this behaviour using the spring.kafka.streams.auto-startup property.

**Additional Kafka Properties**

The properties supported by auto configuration are shown in Common Application properties. Note that, for the most part, these properties (hyphenated or camelCase) map directly to the Apache Kafka dotted properties. Refer to the Apache Kafka documentation for details.

The first few of these properties apply to all components (producers, consumers, admins, and streams) but can be specified at the component level if you wish to use different values. 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 producer or consumer 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
```
This sets the common `prop.one` Kafka property to `first` (applies to producers, consumers and admins), 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**

```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,org.acme
```

**Yaml**

```yaml
spring:
  kafka:
    consumer:
      value-deserializer:
        "org.springframework.kafka.support.serializer.JsonDeserializer"
      properties:
        
```

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

Properties set in this way override any configuration item that Spring Boot explicitly supports.

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");
  }
  ```

- Configure a property name on the @EmbeddedKafka annotation:

  ```java
  @EmbeddedKafka(topics = "someTopic",
                 bootstrapServersProperty = "spring.kafka.bootstrap-servers")
  ```

- Use a placeholder in configuration properties:
4.15. Calling REST Services with 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
@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);
    }
}
```

`RestTemplateBuilder` includes a number of useful methods that can be used to quickly configure a `RestTemplate`. For example, to add BASIC auth support, you can use `builder.basicAuthentication("user", "password").build()`.

4.15.1. 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
static class ProxyCustomizer implements RestTemplateCustomizer {

    @Override
    public void customize(RestTemplate restTemplate) {
        HttpHost proxy = new HttpHost("proxy.example.com");
        HttpClient httpClient = HttpClientBuilder.create().setRoutePlanner(new
        DefaultProxyRoutePlanner(proxy) {
            @Override
            public HttpHost determineProxy(HttpHost target, HttpRequest request,
            HttpContext context) throws HttpException {
                if (target.getHostName().equals("192.168.0.5")) {
                    return null;
                }
                return super.determineProxy(target, request, context);
            }
        }).build();
        restTemplate.setRequestFactory(new
        HttpComponentsClientHttpRequestFactory(httpClient));
    }
}
```

Finally, you can also create your own **RestTemplateBuilder** bean. To prevent switching off the auto-configuration of a **RestTemplateBuilder** and prevent any **RestTemplateCustomizer** beans from being used, make sure to configure your custom instance with a **RestTemplateBuilderConfigurer**. The following example exposes a **RestTemplateBuilder** with what Spring Boot would auto-configure, except that custom connect and read timeouts are also specified:

```java
@Bean
public RestTemplateBuilder restTemplateBuilder(RestTemplateBuilderConfigurer
    configurator) {
    return configurator.configure(new
    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. Doing so switches off the auto-configuration of a `RestTemplateBuilder` and prevents any `RestTemplateCustomizer` beans from being used.

### 4.16. Calling REST Services with 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
@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);
    }
}
```

#### 4.16.1. WebClient Runtime

Spring Boot will auto-detect which `ClientHttpConnector` to use to drive `WebClient`, depending on the libraries available on the application classpath. For now, Reactor Netty and Jetty RS client are supported.

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.

### 4.16.2. 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.

### 4.17. 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 `javax.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
@Service
@Validated
public class MyBean {
    public Archive findByCodeAndAuthor(@Size(min = 8, max = 10) String code, Author author) {
        ...
    }
}
```
4.18. 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.

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**

```properties
spring.mail.properties[mail.smtp.connectiontimeout]=5000
spring.mail.properties[mail.smtp.timeout]=3000
spring.mail.properties[mail.smtp.writetimeout]=5000
```

**Yaml**

```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**

```properties
spring.mail.jndi-name=mail/Session
```

**Yaml**

```yaml
spring:
  mail:
    jndi-name: "mail/Session"
```

When a `jndi-name` is set, it takes precedence over all other Session-related settings.

4.19. Distributed Transactions with JTA

Spring Boot supports distributed JTA transactions across multiple XA resources by using an
Atomikos embedded transaction manager. Deprecated support for using a Bitronix embedded transaction manager is also provided but it will be removed in a future release. JTA transactions are also supported when deploying to a suitable Java EE Application Server.

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.

4.19.1. Using an Atomikos Transaction Manager

Atomikos is a popular open source transaction manager which can be embedded into your Spring Boot application. You can use the spring-boot-starter-jta-atomikos starter to pull in the appropriate Atomikos libraries. Spring Boot auto-configures Atomikos and ensures that appropriate depends-on settings are applied to your Spring beans for correct startup and shutdown ordering.

By default, Atomikos transaction logs are written to a transaction-logs directory in your application’s home directory (the directory in which your application jar file resides). You can customize the location of this directory by setting a spring.jta.log-dir property in your application.properties file. Properties starting with spring.jta.atomikos.properties can also be used to customize the Atomikos UserTransactionServiceImp. See the AtomikosProperties Javadoc for complete details.

To ensure that multiple transaction managers can safely coordinate the same resource managers, each Atomikos instance must be configured with a unique ID. By default, this ID is the IP address of the machine on which Atomikos is running. To ensure uniqueness in production, you should configure the spring.jta.transaction-manager-id property with a different value for each instance of your application.

4.19.2. Using a Bitronix Transaction Manager

As of Spring Boot 2.3, support for Bitronix has been deprecated and will be removed in a future release.

You can use the spring-boot-starter-jta-bitronix starter to add the appropriate Bitronix dependencies to your project. As with Atomikos, Spring Boot automatically configures Bitronix and post-processes your beans to ensure that startup and shutdown ordering is correct.

By default, Bitronix transaction log files (part1.btm and part2.btm) are written to a transaction-logs directory in your application home directory. You can customize the location of this directory by setting the spring.jta.log-dir property. Properties starting with spring.jta.bitronix.properties are also bound to the bitronix.tm.Configuration bean, allowing for complete customization. See the Bitronix documentation for details.
To ensure that multiple transaction managers can safely coordinate the same resource managers, each Bitronix instance must be configured with a unique ID. By default, this ID is the IP address of the machine on which Bitronix is running. To ensure uniqueness in production, you should configure the `spring.jta.transaction-manager-id` property with a different value for each instance of your application.

### 4.19.3. Using a Java EE Managed Transaction Manager

If you package your Spring Boot application as a `war` or `ear` file and deploy it to a Java 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). If you use 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`.

### 4.19.4. Mixing XA and Non-XA JMS Connections

When using JTA, the primary JMS `ConnectionFactory` bean is XA-aware and participates in distributed transactions. 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 inject the `nonXaJmsConnectionFactory` bean rather than the `@Primary jmsConnectionFactory` bean. For consistency, the `jmsConnectionFactory` bean is also provided by using the bean alias `xaJmsConnectionFactory`.

The following example shows how to inject `ConnectionFactory` instances:

```java
// Inject the primary (XA aware) ConnectionFactory
@Autowired
private ConnectionFactory defaultConnectionFactory;

// Inject the XA aware ConnectionFactory (uses the alias and injects the same as above)
@Autowired
@Qualifier("xaJmsConnectionFactory")
private ConnectionFactory xaConnectionFactory;

// Inject the non-XA aware ConnectionFactory
@Autowired
@Qualifier("nonXaJmsConnectionFactory")
private ConnectionFactory nonXaConnectionFactory;
```
4.19.5. Supporting an Alternative Embedded Transaction Manager

The **XAConnectionFactoryWrapper** and **XADataSourceWrapper** interfaces can be used to support alternative 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**.

The **AtomikosXAConnectionFactoryWrapper** and **AtomikosXADatasourceWrapper** provide good examples of how to write XA wrappers.

4.20. 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** in the working directory or at the root of the classpath.

Spring Boot supports both Hazelcast 4 and Hazelcast 3. If you downgrade to Hazelcast 3, **hazelcast-client** should be added to the classpath to configure a client.

If a client can't 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**

```properties
spring.hazelcast.config=classpath:config/my-hazelcast.xml
```

**Yaml**

```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.

Spring Boot also has explicit caching support for Hazelcast. If caching is enabled, the HazelcastInstance is automatically wrapped in a CacheManager implementation.

### 4.21. 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**

```properties
spring.quartz.job-store-type=jdbc
```

**Yaml**

```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**

```properties
spring.quartz.jdbc.initialize-schema=always
```

**Yaml**

```yaml
spring:
  quartz:
    jdbc:
      initialize-schema: "always"
```
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.*`.

In particular, an `Executor` bean is not associated with the scheduler as Quartz offers a way to configure the scheduler via `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
public class SampleJob extends QuartzJobBean {

    private MyService myService;

    private String name;

    // Inject "MyService" bean
    public void setMyService(MyService myService) { ... }

    // Inject the "name" job data property
    public void setName(String name) { ... }

    @Override
    protected void executeInternal(JobExecutionContext context)
        throws JobExecutionException {
            ...
    }
}
```
4.22. 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.

If you have defined a custom Executor in the context, regular task execution (i.e. @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

```properties
spring.task.execution.pool.max-size=16
spring.task.execution.pool.queue-capacity=100
spring.task.execution.pool.keep-alive=10s
```

Yaml

```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 (@EnableScheduling). The thread pool uses one thread by default and those settings can be fine-tuned using the spring.task.scheduling namespace.

Both a TaskExecutorBuilder bean and a TaskSchedulerBuilder bean are made available in the context if a custom executor or scheduler needs to be created.
4.23. 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 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

```properties
spring.integration.jdbc.initialize-schema=always
```

Yaml

```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

```properties
# Connecting to a RSocket server over TCP
spring.integration.rsocket.client.host=example.org
spring.integration.rsocket.client.port=9898
```

Yaml

```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:
        url: "ws://example.org"

See the IntegrationAutoConfiguration and IntegrationProperties classes for more details.

By default, if a Micrometer meterRegistry bean is present, Spring Integration metrics will be managed by Micrometer. If you wish to use legacy Spring Integration metrics, add a DefaultMetricsFactory bean to the application context.

4.24. 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:

- JDBC
- Redis
- Hazelcast
- MongoDB

The Servlet auto-configuration replaces the need to use @Enable*HttpSession.

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.

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, you must choose the StoreType that you wish to use to store the sessions. For instance, to use JDBC as the back-end store, you can configure your application as follows:

Properties

spring.session.store-type=jdbc
You can disable Spring Session by setting the `store-type` to `none`.

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.

### 4.25. Monitoring and Management over JMX

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. 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 will use that and default to the VM `MBeanServer` if necessary. If all that fails, a new `MBeanServer` will be created.

See the `JmxAutoConfiguration` class for more details.

### 4.26. 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.

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:

```xml
<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>
```

hamster-core is excluded in favor of org.hamcrest:hamcrest that is part of spring-boot-starter-test.

### 4.26.1. 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.

### 4.26.2. 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.

4.26.3. 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.

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.

If you are using JUnit 4, don't forget to also add @RunWith(SpringRunner.class) to your test, otherwise the annotations will be ignored. If you are using JUnit 5, there's 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).
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.

`@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’ll 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
@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.

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
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.

**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. When placed on a top-level class, `@TestConfiguration` indicates that classes in `src/test/java` should not be picked up by scanning. You can then import that class explicitly where it is required, as shown in the following example:

```java
@SpringBootTest
@Import(MyTestsConfiguration.class)
class MyTests {

    @Test
    void exampleTest() {
        ...
    }
}
```

If you directly use `@ComponentScan` (that is, not through `@SpringBootApplication`) you need to register the `TypeExcludeFilter` with it. See the Javadoc for details.

**Using Application Arguments**

If your application expects arguments, you can have `@SpringBootTest` inject them using the `args` attribute.

```java
@SpringBootTest(args = "--app.test=one")
class ApplicationArgumentsExampleTests {

    @Test
    void applicationArgumentsPopulated(@Autowired ApplicationArguments args) {
        assertThat(args.getOptionNames()).containsOnly("app.test");
        assertThat(args.getOptionValues("app.test")).containsOnly("one");
    }
}
```
Testing with a mock environment
By default, @SpringBootTest does not start the server. If you have web endpoints that you want to
test against this mock environment, you can additionally configure MockMvc as shown in the
following example:

import org.junit.jupiter.api.Test;
import
import
import
import

org.springframework.beans.factory.annotation.Autowired;
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
org.springframework.boot.test.context.SpringBootTest;
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 MockMvcExampleTests {
@Test
void exampleTest(@Autowired MockMvc mvc) throws Exception {
mvc.perform(get("/")).andExpect(status().isOk()).andExpect(content().string("Hello
World"));
}
}



If you want to focus only on the web layer and not start a complete
ApplicationContext, consider using @WebMvcTest instead.

Alternatively, you can configure a WebTestClient as shown in the following example:

209


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.

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 @Autowired 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:
This setup requires spring-webflux on the classpath. If you can’t or won’t 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 RandomPortTestRestTemplateExampleTests {

    @Test
    void exampleTest(@Autowired TestRestTemplate restTemplate) {
        String body = restTemplate.getForObject("/", String.class);
        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
@ExtendWith(SpringExtension.class)
@SpringBootTest(properties = "spring.jmx.enabled=true")
@DirtiesContext
class SampleJmxTests {

    @Autowired
    private MBeanServer mBeanServer;

    @Test
    void exampleTest() {
        // ...
    }
}
```

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 `@AutoConfigureMetrics`.

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.

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, a listener must be explicitly added, as shown in the following example:

```java
@TestExecutionListeners(MockitoTestExecutionListener.class)
```
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.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.mock.mockito.*;
import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@SpringBootTest
class MyTests {
    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Reverser reverser;

    @Test
    void exampleTest() {
        // RemoteService has been injected into the reverser bean
        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }
}
```

@MockBean cannot be used to mock the behavior of a bean that’s 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.

CGLib proxies, such as those created for scoped beans, declare the proxied methods as final. This stops Mockito from functioning correctly as it cannot mock or spy on final methods in its default configuration. If you want to mock or spy on such a bean, configure Mockito to use its inline mock maker by adding org.mockito:mockito-inline to your application’s test dependencies. This allows Mockito to mock and spy on final methods.
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.

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.

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.

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`.

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.

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 Modules
- Gson
- Jsonb

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:
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.json.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.json.*;
import static org.assertj.core.api.Assertions.*;

@JsonTest
class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    void testSerialize() 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 testDeserialize() 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");
    }
}

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're using 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.
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`, and `HandlerMethodArgumentResolver`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@WebMvcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

- A list of the auto-configuration settings that are enabled by `@WebMvcTest` can be found in the appendix.

- 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.

- You can also auto-configure `MockMvc` in a non-`@WebMvcTest` (such as `@SpringBootTest`) by annotating it with `@AutoConfigureMockMvc`. The following example uses MockMvc:

```java
assertThat(json.write(message))
    .extractingJsonPathNumberValue("@.test.numberValue")
    .satisfies((number) -> assertThat(number.floatValue()).isCloseTo(0.15f, within(0.01f)));
```
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 or Selenium, auto-configuration also provides an HtmlUnit WebClient bean and/or a Selenium WebDriver bean. The following example uses HtmlUnit:
import com.gargoylesoftware.htmlunit.*;
import org.junit.jupiter.api.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;
import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@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");
    }
}

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.

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.

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.

A list of the auto-configureations that are enabled by `@WebFluxTest` can be found in the appendix.

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.

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.http.MediaType;
import org.springframework.test.web.reactive.server.WebTestClient;

@WebFluxTest(UserVehicleController.class)
class MyControllerTests {
    @Autowired
    private WebTestClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        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");
    }
}
```
This setup is only supported by WebFlux applications as using `WebTestClient` in a mocked web application only works with WebFlux at the moment.

@WebFluxTest cannot detect routes registered via the functional web framework. For testing `RouterFunction` beans in the context, consider importing your `RouterFunction` yourself via `@Import` or using `@SpringBootTest`.

@WebFluxTest cannot detect custom security configuration registered via a `@Bean` of type `SecurityWebFilterChain`. To include that in your test, you will need to import the configuration that registers the bean via `@Import` or use `@SpringBootTest`.

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 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", earlier in this chapter.)

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 ExampleDataCassandraTests {
    @Autowired
    private YourRepository repository;

    //
}
```

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. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataJpaTest` annotation is used. `@EnableConfigurationProperties` can be used to
include `@ConfigurationProperties` beans.

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.junit.jupiter.api.Test;
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 ExampleNonTransactionalTests {
}
```

Data JPA tests may also inject a `TestEntityManager` bean, which provides an alternative to the standard JPA `EntityManager` that is specifically designed for tests. If you want to use `TestEntityManager` outside of `@DataJpaTest` instances, you can also use the `@AutoConfigureTestEntityManager` annotation. A `JdbcTemplate` is also available if you need that. The following example shows the `@DataJpaTest` annotation in use:
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.orm.jpa.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest.

@DataJpaTest
class ExampleRepositoryTests {
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    void testExample() throws Exception {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getVin()).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:

@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
class ExampleRepositoryTests {
    // ...
}

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.

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.junit.jupiter.api.Test;
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 ExampleNonTransactionalTests {
}
```

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. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataJdbcTest` annotation is used. `@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 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”, earlier in this chapter.) 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.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;

@JooqTest
class ExampleJooqTests {
    @Autowired
    private 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 an in-memory embedded MongoDB (if available), 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", earlier in this chapter.)

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 ExampleDataMongoTests {
    @Autowired
    private MongoTemplate mongoTemplate;
    //
}
```

In-memory embedded MongoDB 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 MongoDB server, you should exclude the embedded MongoDB auto-configuration, as shown in the following example:
import org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;

@DataMongoTest(excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class)
class ExampleDataMongoNonEmbeddedTests {
}

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", earlier in this chapter.)

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:

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;

@DataNeo4jTest
class ExampleDataNeo4jTests {
    
    @Autowired
    private YourRepository repository;

    //
}

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:
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 ExampleNonTransactionalTests {
    
}

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", earlier in this chapter.)

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 ExampleDataRedisTests {
    @Autowired
    private YourRepository repository;

    //
}
```

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", earlier in this chapter.)
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 ExampleDataLdapTests {
    @Autowired
    private 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 ExampleDataLdapNonEmbeddedTests {
}
```

### 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.

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:
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:
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
@TestConfiguration
static class CustomizationConfiguration
    implements RestDocsMockMvcConfigurationCustomizer {

    @Override
    public void customize(MockMvcRestDocumentationConfigurer configurer) {
        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:
@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:

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:
@TestConfiguration(proxyBeanMethods = false)
public static class CustomizationConfiguration implements
RestDocsWebTestClientConfigurationCustomizer {

    @Override
    public void customize(WebTestClientRestDocumentationConfigurer configurer) {
        configurer.snippets().withEncoding("UTF-8");
    }
}

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.web.server.LocalServerPort;
import static io.restassured.RestAssured.given;
import static org.hamcrest.Matchers.is;
import static org.springframework.restdocs.restassured3.RestAssuredRestDocumentation.document;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
class UserDocumentationTests {

    @Test
    void listUsers(@Autowired RequestSpecification documentationSpec, @LocalServerPort int port) {
        given(documentationSpec).filter(document("list-users")).when().port(port).get("/").then().assertThat()
            .statusCode(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:
Auto-configuration Settings

Each slice provides one or more `@AutoConfigure`... annotations that define 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:
@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration.class)
class ExampleJdbcTests {
}

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 META-INF/spring.factories as shown in the following example:

```
org.springframework.boot.test.autoconfigure.jdbc.JdbcTest=com.example.IntegrationAutoConfiguration
```

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 Batch and you rely on the auto-configuration for it. You could define your @SpringBootApplication as follows:

```
@SpringBootApplication
@EnableBatchProcessing
public class SampleApplication { ... }
```

Because this class is the source configuration for the test, any slice test actually tries to start Spring Batch, 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:

```
@Configuration(proxyBeanMethods = false)
@EnableBatchProcessing
public class BatchConfiguration { ... }
```
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.

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
@Configuration
public class WebConfiguration {
    @Bean
    public WebMvcConfigurer testConfigurer() {
        return new WebMvcConfigurer() {
            ...
        };
    }
}
```

The configuration below will, however, cause the custom `WebMvcConfigurer` to be loaded by the test slice.

```java
@Component
public class TestWebMvcConfigurer implements 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
@SpringBootApplication
@ComponentScan({ "com.example.app", "org.acme.another" })
public class SampleApplication { ...
}
```

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.

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

If you wish to use Spock to test a Spring Boot application, you should add a dependency on Spock's spock-spring module to your application's build. spock-spring integrates Spring's test framework into Spock. It is recommended that you use Spock 1.2 or later to benefit from a number of improvements to Spock's Spring Framework and Spring Boot integration. See the documentation for Spock's Spring module for further details.

4.26.4. Test Utilities

A few test utility classes that are generally useful when testing your application are packaged as part of spring-boot.

ConfigFileApplicationContextInitializer

ConfigFileApplicationContextInitializer 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
@ContextConfiguration(classes = Config.class,
                        initializers = ConfigFileApplicationContextInitializer.class)
```

Using ConfigFileApplicationContextInitializer 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:

```
TestPropertyValues.of("org=Spring", "name=Boot").applyTo(env);
```

OutputCapture

OutputCapture is a JUnit Extension that you can use to capture System.out and System.err output. To use add @ExtendWith(OutputCaptureExtension.class) and inject CapturedOutput as an argument to your test class constructor or test method as follows:
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 behaves in a test-friendly way by not throwing exceptions on server-side errors.

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 4.3.2 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
public class MyTest {

    private TestRestTemplate template = new TestRestTemplate();

    @Test
    public void testRequest() throws Exception {
        HttpHeaders headers = this.template.getForEntity("https://myhost.example.com/example", String.class).getHeaders();
        assertThat(headers.getLocation()).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
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class SampleWebClientTests {

    @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 Config {

        @Bean
        RestTemplateBuilder restTemplateBuilder() {
            return new RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1))
                .setReadTimeout(Duration.ofSeconds(1));
        }
    }
}
```

### 4.27. 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:

```xml
<dependency>
    <groupId>javax.websocket</groupId>
    <artifactId>javax.websocket-api</artifactId>
</dependency>
```
4.28. 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

```properties
spring.webservices.wsdl-locations=classpath:/wsdl
```

Yaml

```yaml
spring:
  webservices:
    wsdl-locations: "classpath:/wsdl"
```

4.28.1. 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
@Service
public class MyService {

  private final WebServiceTemplate webServiceTemplate;

  public MyService(WebServiceTemplateBuilder webServiceTemplateBuilder) {
    this.webServiceTemplate = webServiceTemplateBuilder.build();
  }

  public DetailsResp someWsCall(DetailsReq detailsReq) {
    return (DetailsResp) this.webServiceTemplate.marshalSendAndReceive(detailsReq, new SoapActionCallback(ACTION));
  }
}
```
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
@Bean
public WebServiceTemplate webServiceTemplate(WebServiceTemplateBuilder builder) {
    return builder.messageSenders(new HttpWebServiceMessageSenderBuilder()
}
```

### 4.29. 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.

A demo project is available to showcase how you can create a starter step-by-step.

#### 4.29.1. Understanding Auto-configured Beans

Under the hood, auto-configuration is implemented with 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 `@Configuration` classes that Spring provides (see the `META-INF/spring.factories` file).

#### 4.29.2. Locating Auto-configuration Candidates

Spring Boot checks for the presence of a `META-INF/spring.factories` file within your published jar. The file should list your configuration classes under the `EnableAutoConfiguration` key, as shown in the following example:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=
    com.mycorp.libx.autoconfigure.LibXAutoConfiguration,
    com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```
Auto-configurations must be loaded that way only. 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` should be used instead.

You can use the `@AutoConfigureAfter` or `@AutoConfigureBefore` annotations if your configuration needs to be applied in a specific order. 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.

### 4.29.3. 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
@Configuration(proxyBeanMethods = false)
// Some conditions
public class MyAutoConfiguration {

    // Auto-configured beans

    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(EmbeddedAcmeService.class)
    static class EmbeddedConfiguration {

        @Bean
        @ConditionalOnMissingBean
        public EmbeddedAcmeService embeddedAcmeService() { ... }

    }

}
```

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
@Configuration(proxyBeanMethods = false)
public class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public MyService myService() { ... }

}
```

In the preceding example, the `myService` bean is going to be created if no bean of type `MyService` is already contained in the `ApplicationContext`. 
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).

`@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.

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’s 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` annotation lets configuration be included depending on whether the application is a traditional WAR application that is deployed to a container. This condition will not match for applications that are run with an embedded server.

**SpEL Expression Conditions**

The `@ConditionalOnExpression` annotation lets configuration be included based on the result of a SpEL expression.
4.29.4. 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.

ApplicationContextRunner is usually defined as a field of the test class to gather the base, common configuration. The following example makes sure that UserServiceAutoConfiguration is always invoked:

```java
private final ApplicationContextRunner contextRunner = new ApplicationContextRunner()
    .withConfiguration(AutoConfigurations.of(UserServiceAutoConfiguration.class));
```

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(UserService.class);
        assertThat(context).getBean("myUserService").isSameAs(context.getBean(UserService.class));
    });
}

@Configuration(proxyBeanMethods = false)
static class UserConfiguration {
    @Bean
    UserService myUserService() {
        return new UserService("mine");
    }
}
```

It is also possible to easily customize the Environment, as shown in the following example:
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
@Test
public void autoConfigTest {
    ConditionEvaluationReportLoggingListener initializer = new
    ConditionEvaluationReportLoggingListener(
        LogLevel.INFO);
    ApplicationContextRunner contextRunner = new ApplicationContextRunner()
        .withInitializer(initializer).run((context) -> {
            // Do something...
        });
}
```

### 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 `UserService` is not present, the auto-configuration is properly disabled:

```java
@Test
void serviceIsIgnoredIfLibraryIsNotPresent() {
    this.contextRunner.withClassLoader(new FilteredClassLoader(UserService.class))
        .run((context) -> assertThat(context).doesNotHaveBean("userService"));
}
```

### 4.29.5. 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 flavours, 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 feature, 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 (e.g. `acme`).

Make sure that configuration keys are documented by adding field javadoc for each property, as shown in the following example:
@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);

    // getters & setters
}

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, e.g. "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.

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. It is recommended to add the following dependency in a module that contains auto-configurations:

```xml
<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:

```xml
<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 4.5 and earlier, the dependency should be declared in the `compileOnly` configuration, as shown in the following example:

```groovy
dependencies {
    compileOnly "org.springframework.boot:spring-boot-autoconfigure-processor"
}
```

With Gradle 4.6 and later, the dependency should be declared in the `annotationProcessor` configuration, as shown in the following example:

```groovy
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.

Either way, your starter must reference the core Spring Boot starter (spring-boot-starter) directly or indirectly (i.e. 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.

4.30. 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 via 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.

4.30.1. Requirements

Spring Boot supports Kotlin 1.3.x. 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.

These dependencies and plugins are provided by default if one bootstraps a Kotlin project on start.spring.io.

4.30.2. 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 via 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).

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.

4.30.3. Kotlin API

runApplication

Spring Boot provides an idiomatic way to run an application with runApplication<MyApplication>(*args) as shown in the following example:

```kotlin
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:

```kotlin
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.

### 4.30.4. 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 via 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 via the `kotlin-coroutines.version` property.

```kotlin
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`.
```

### 4.30.5. @ConfigurationProperties

@`ConfigurationProperties` when used in combination with `@ConstructorBinding` supports classes with immutable `val` properties as shown in the following example:

```kotlin
@ConstructorBinding
@ConfigurationProperties("example.kotlin")
data class KotlinExampleProperties(
    val name: String,
    val description: String,
    val myService: MyService) {

    data class MyService(
        val apiToken: String,
        val uri: URI
    )
}
```

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.
4.30.6. 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.

4.30.7. Resources

Further reading

- Kotlin language reference
- Kotlin Slack (with a dedicated #spring channel)
- Stackoverflow 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

4.31. 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.

### 4.31.1. 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:

```plaintext
- "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.

For Maven, refer to the [packaging layered jars section](#) for more details on adding a layer index to the jar. For Gradle, refer to the [packaging layered jars section](#) of the Gradle plugin documentation.

### 4.31.2. Building Container Images

Spring Boot applications can be containerized using Dockerfiles, or by using Cloud Native Buildpacks to create docker compatible container images that you can run anywhere.

**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.

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:

```bash
$ 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's an example of a Dockerfile using `jarmode`.

```Dockerfile
FROM adoptopenjdk:11-jre-hotspot as builder
WORKDIR application
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layertools -jar application.jar extract

FROM adoptopenjdk:11-jre-hotspot
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:

```bash
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.

**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.

Refer to the individual plugin documentation on how to use buildpacks with Maven and Gradle.

The Paketo Spring Boot buildpack has also been updated to support the `layers.idx` file so any customization that is applied to it will be reflected in the image created by the buildpack.

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`.

### 4.32. What to Read Next

If you want to learn more about any of the classes discussed in this section, you can check out the Spring Boot API documentation or you can browse the source code directly. If you have specific questions, take a look at 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 5. Spring Boot Actuator: 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.

5.1. 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:

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

For Gradle, use the following declaration:

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

5.2. 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.

Each individual endpoint can be enabled or disabled and exposed (made remotely accessible) over HTTP or JMX. An endpoint is considered to be available when it is both enabled and exposed. The built-in endpoints will only be auto-configured when they are available. Most applications choose
exposure via HTTP, where the ID of the endpoint along with a prefix of /actuator is mapped to a URL. For example, by default, the health endpoint is mapped to /actuator/health.

The following technology-agnostic endpoints are available:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auditevents</td>
<td>Exposes audit events information for the current application. Requires an AuditEventRepository bean.</td>
</tr>
<tr>
<td>beans</td>
<td>Displays a complete list of all the Spring beans in your application.</td>
</tr>
<tr>
<td>caches</td>
<td>Exposes available caches.</td>
</tr>
<tr>
<td>conditions</td>
<td>Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.</td>
</tr>
<tr>
<td>configprops</td>
<td>Displays a collated list of all @ConfigurationProperties.</td>
</tr>
<tr>
<td>env</td>
<td>Exposes properties from Spring’s ConfigurableEnvironment.</td>
</tr>
<tr>
<td>flyway</td>
<td>Shows any Flyway database migrations that have been applied. Requires one or more Flyway beans.</td>
</tr>
<tr>
<td>health</td>
<td>Shows application health information.</td>
</tr>
<tr>
<td>httptrace</td>
<td>Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges). Requires an HttpTraceRepository bean.</td>
</tr>
<tr>
<td>info</td>
<td>Displays arbitrary application info.</td>
</tr>
<tr>
<td>integrationgraph</td>
<td>Shows the Spring Integration graph. Requires a dependency on spring-integration-core.</td>
</tr>
<tr>
<td>loggers</td>
<td>Shows and modifies the configuration of loggers in the application.</td>
</tr>
<tr>
<td>liquibase</td>
<td>Shows any Liquibase database migrations that have been applied. Requires one or more Liquibase beans.</td>
</tr>
<tr>
<td>metrics</td>
<td>Shows 'metrics' information for the current application.</td>
</tr>
<tr>
<td>mappings</td>
<td>Displays a collated list of all @RequestMapping paths.</td>
</tr>
<tr>
<td>scheduledtasks</td>
<td>Displays the scheduled tasks in your application.</td>
</tr>
<tr>
<td>sessions</td>
<td>Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Requires a Servlet-based web application using Spring Session.</td>
</tr>
<tr>
<td>shutdown</td>
<td>Lets the application be gracefully shutdown. Disabled by default.</td>
</tr>
<tr>
<td>startup</td>
<td>Shows the startup steps data collected by the ApplicationStartup. Requires the SpringApplication to be configured with a BufferingApplicationStartup.</td>
</tr>
<tr>
<td>threaddump</td>
<td>Performs a thread dump.</td>
</tr>
</tbody>
</table>

If your application is a web application (Spring MVC, Spring WebFlux, or Jersey), you can use the following additional endpoints:
### ID | Description
---|---
heapdump | Returns an hprof heap dump file.
jolokia | Exposes JMX beans over HTTP (when Jolokia is on the classpath, not available for WebFlux). Requires a dependency on jolokia-core.
logfile | Returns the contents of the logfile (if logging.file.name or logging.file.path properties have 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.

To learn more about the Actuator’s endpoints and their request and response formats, please refer to the separate API documentation (HTML or PDF).

#### 5.2.1. 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**

```properties
management.endpoint.shutdown.enabled=true
```

**Yaml**

```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**

```properties
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
```
Yaml

```yaml
management:
  endpoints:
    enabled-by-default: false
  endpoint:
    info:
      enabled: true
```

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.

### 5.2.2. Exposing Endpoints

Since Endpoints may contain sensitive information, careful consideration should be given about when to expose them. The following table shows the default exposure for the built-in endpoints:

<table>
<thead>
<tr>
<th>ID</th>
<th>JMX</th>
<th>Web</th>
</tr>
</thead>
<tbody>
<tr>
<td>auditevents</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>beans</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>caches</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>conditions</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>configprops</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>env</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>flyway</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>health</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>heapdump</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>httptrace</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>info</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>integrationgraph</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>jolokia</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>logfile</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>loggers</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>liquibase</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>metrics</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>mappings</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>prometheus</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>scheduledtasks</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ID</td>
<td>JMX</td>
<td>Web</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>sessions</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>shutdown</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>start up</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>thread dump</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

To change which endpoints are exposed, use the following technology-specific `include` and `exclude` properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>management.endpoints.jmx.exposure.exclude</code></td>
<td></td>
</tr>
<tr>
<td><code>management.endpoints.jmx.exposure.include</code></td>
<td>*</td>
</tr>
<tr>
<td><code>management.endpoints.web.exposure.exclude</code></td>
<td></td>
</tr>
<tr>
<td><code>management.endpoints.web.exposure.include</code></td>
<td><code>info, health</code></td>
</tr>
</tbody>
</table>

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. Both `include` and `exclude` properties can be configured with a list of endpoint IDs.

For example, to stop exposing all endpoints over JMX and only expose the `health` and `info` endpoints, use the following property:

**Properties**

```properties
management.endpoints.jmx.exposure.include=health,info
```

**Yaml**

```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**

```properties
management.endpoints.web.exposure.include=*  
management.endpoints.web.exposure.exclude=env,beans
```
5.2.3. Securing HTTP Endpoints

You should take care to secure HTTP endpoints in the same way that you would any other sensitive URL. If Spring Security is present, endpoints are secured by default using Spring Security’s content-negotiation strategy. If you wish to configure custom security for HTTP endpoints, for example, only allow users with a certain role to access them, Spring Boot provides some convenient RequestMatcher objects that can be used in combination with Spring Security.

A typical Spring Security configuration might look something like the following example:

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests((requests) ->
        requests.anyRequest().hasRole("ENDPOINT_ADMIN"));
    http.httpBasic();
    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:
Additionally, if Spring Security is present, you would need to add custom security configuration that allows unauthenticated access to the endpoints as shown in the following example:

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests((requests) ->
            requests.anyRequest().permitAll());
    return http.build();
}
```

In both the examples above, 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 will need to configure an additional `SecurityFilterChain` bean with rules that apply to the rest of the application.

### 5.2.4. 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 will cache 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**

```yaml
management:
    endpoint:
        beans:
            cache:
                time-to-live: "10s"
```
The prefix `management.endpoint.<name>` is used to uniquely identify the endpoint that is being configured.

5.2.5. 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.

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`, then 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.

5.2.6. 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, Actuator's web endpoints can be configured to support such scenarios.

CORS support is disabled by default and is only enabled once the `management.endpoints.web.cors.allowed-origins` property has been set. The following configuration permits GET and POST calls from the example.com domain:

**Properties**

```properties
management.endpoints.web.cors.allowed-origins=https://example.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

**Yaml**

```yaml
management:
  endpoints:
    web:
      cors:
        allowed-origins: "https://example.com"
        allowed-methods: "GET,POST"
```

See [CorsEndpointProperties](#) for a complete list of options.

5.2.7. 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 using Jersey, Spring MVC, or Spring WebFlux. If both Jersey and Spring MVC are available, Spring MVC will be used.

The following example exposes a read operation that returns a custom object:
@ReadOperation
def getCustomData() {
    return new 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 via their parameters. When exposed via the web, the values for these parameters are taken from the URL's query parameters and from the JSON request body. When exposed via 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.

Each root property in the JSON request body can be mapped to a parameter of the endpoint. Consider the following JSON request body:

```json
{
    "name": "test",
    "counter": 42
}
```

This can be used to invoke a write operation that takes `String name` and `int counter` parameters, as shown in the following example:

```java
@WriteOperation
def updateCustomData(String name, int counter) {
    // injects "test" and 42
}
```

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 defining a `name` and `counter` properties is not supported.
To allow the input to be mapped to the operation method’s parameters, Java code implementing an endpoint should be compiled with `-parameters`, and Kotlin code implementing an endpoint should be compiled with `-java-parameters`. This will happen automatically if you are using Spring Boot's Gradle plugin or if you are using 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 via JMX or an HTTP request is converted to the required types 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 will be 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 web-exposed endpoints. The default base path is `/actuator`. For example, an endpoint with the ID `sessions` will use `/actuator/sessions` as its path in the predicate.

The path can be further customized 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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>HTTP method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@ReadOperation</code></td>
<td>GET</td>
</tr>
<tr>
<td><code>@WriteOperation</code></td>
<td>POST</td>
</tr>
<tr>
<td><code>@DeleteOperation</code></td>
<td>DELETE</td>
</tr>
</tbody>
</table>

**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.

**Produce**

The produces clause of the predicate can be determined by the `provides` 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.

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 will not be called and the response status will be 400 (Bad Request).

**Web Endpoint Range Requests**

An HTTP range request can be used 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.

> 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 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

@ControllerEndpoint and @RestControllerEndpoint can be used to implement an endpoint that is only exposed by Spring MVC or Spring WebFlux. Methods are mapped 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.

5.2.8. 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>never</td>
<td>Details are never shown.</td>
</tr>
<tr>
<td>when-authorized</td>
<td>Details are only shown to authorized users. Authorized roles can be configured using management.endpoint.health.roles.</td>
</tr>
<tr>
<td>always</td>
<td>Details are shown to all users.</td>
</tr>
</tbody>
</table>

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. The roles can be configured using the management.endpoint.health.roles property.

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 either be 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.
The HealthContributorRegistry can be used to register and unregister health indicators at runtime.

Auto-configured HealthIndicators

The following HealthIndicators are auto-configured by Spring Boot when appropriate. You can also enable/disable selected indicators by configuring management.health.key.enabled, with the key listed in the table below.

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassandra</td>
<td>CassandraDriverHealthIndicator</td>
<td>Checks that a Cassandra database is up.</td>
</tr>
<tr>
<td>couchbase</td>
<td>CouchbaseHealthIndicator</td>
<td>Checks that a Couchbase cluster is up.</td>
</tr>
<tr>
<td>datasource</td>
<td>DataSourceHealthIndicator</td>
<td>Checks that a connection to DataSource can be obtained.</td>
</tr>
<tr>
<td>diskspace</td>
<td>DiskSpaceHealthIndicator</td>
<td>Checks for low disk space.</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>ElasticsearchRestHealthIndicator</td>
<td>Checks that an Elasticsearch cluster is up.</td>
</tr>
<tr>
<td>hazelcast</td>
<td>HazelcastHealthIndicator</td>
<td>Checks that a Hazelcast server is up.</td>
</tr>
<tr>
<td>influxdb</td>
<td>InfluxDbHealthIndicator</td>
<td>Checks that an InfluxDB server is up.</td>
</tr>
<tr>
<td>jms</td>
<td>JmsHealthIndicator</td>
<td>Checks that a JMS broker is up.</td>
</tr>
<tr>
<td>ldap</td>
<td>LdapHealthIndicator</td>
<td>Checks that an LDAP server is up.</td>
</tr>
<tr>
<td>mail</td>
<td>MailHealthIndicator</td>
<td>Checks that a mail server is up.</td>
</tr>
<tr>
<td>mongo</td>
<td>MongoHealthIndicator</td>
<td>Checks that a Mongo database is up.</td>
</tr>
<tr>
<td>neo4j</td>
<td>Neo4jHealthIndicator</td>
<td>Checks that a Neo4j database is up.</td>
</tr>
<tr>
<td>ping</td>
<td>PingHealthIndicator</td>
<td>Always responds with UP.</td>
</tr>
<tr>
<td>rabbit</td>
<td>RabbitHealthIndicator</td>
<td>Checks that a Rabbit server is up.</td>
</tr>
<tr>
<td>redis</td>
<td>RedisHealthIndicator</td>
<td>Checks that a Redis server is up.</td>
</tr>
<tr>
<td>solr</td>
<td>SolrHealthIndicator</td>
<td>Checks that a Solr server is up.</td>
</tr>
</tbody>
</table>

You can disable them all by setting the management.health.defaults.enabled property.

Additional HealthIndicators are available but not enabled by default:

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>livenessstate</td>
<td>LivenessStateHealthIndicator</td>
<td>Exposes the &quot;Liveness&quot; application availability state.</td>
</tr>
<tr>
<td>readinessstate</td>
<td>ReadinessStateHealthIndicator</td>
<td>Exposes the &quot;Readiness&quot; application availability state.</td>
</tr>
</tbody>
</table>
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(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }
}
```

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`.

In addition to Spring Boot's predefined `Status` types, it is also possible for `Health` to return a custom `Status` that represents a new system state. In such cases, a custom implementation of the `StatusAggregator` interface also needs to be provided, or the default implementation has to be configured by using the `management.endpoint.health.status.order` configuration property.

For example, assume a new `Status` with code `FATAL` is being used in one of your `HealthIndicator` implementations. To configure the severity order, add the following property to your application properties:

```java
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 they must be configured explicitly 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

```yaml
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
```

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:

<table>
<thead>
<tr>
<th>Status</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOWN</td>
<td>SERVICE_UNAVAILABLE (503)</td>
</tr>
<tr>
<td>OUT_OF_SERVICE</td>
<td>SERVICE_UNAVAILABLE (503)</td>
</tr>
<tr>
<td>UP</td>
<td>No mapping by default, so http status is 200</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>No mapping by default, so http status is 200</td>
</tr>
</tbody>
</table>

**Reactive Health Indicators**

For reactive applications, such as those using 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.

In a reactive application, the ReactiveHealthContributorRegistry should be used to register and unregister health indicators at runtime. If you need to register a regular HealthContributor, you should wrap it using 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
@Component
public class MyReactiveHealthIndicator implements ReactiveHealthIndicator {
    @Override
    public Mono<Health> health() {
        return doHealthCheck() //perform some specific health check that returns a
        .onErrorResume(ex -> Mono.just(new Health.Builder().down(ex).build()));
    }
}
```

To handle the error automatically, consider extending from AbstractReactiveHealthIndicator.

### Auto-configured ReactiveHealthIndicators

The following ReactiveHealthIndicators are auto-configured by Spring Boot when appropriate:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CassandraDriverReactiveHealthIndicator</td>
<td>Checks that a Cassandra database is up.</td>
</tr>
<tr>
<td>CouchbaseReactiveHealthIndicator</td>
<td>Checks that a Couchbase cluster is up.</td>
</tr>
<tr>
<td>ElasticsearchReactiveHealthIndicator</td>
<td>Checks that an Elasticsearch cluster is up.</td>
</tr>
<tr>
<td>MongoReactiveHealthIndicator</td>
<td>Checks that a Mongo database is up.</td>
</tr>
<tr>
<td>Neo4jReactiveHealthIndicator</td>
<td>Checks that a Neo4j database is up.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>RedisReactiveHealthIndicator</td>
<td>Checks that a Redis server is up.</td>
</tr>
</tbody>
</table>

If necessary, reactive indicators replace the regular ones. Also, any `HealthIndicator` that is not handled explicitly is wrapped automatically.

### Health Groups

It's sometimes useful to organize health indicators into groups that can be used 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**

```properties
management.endpoint.health.group.custom.include=db
```

**Yaml**

```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**

```properties
management.endpoint.health.group.custom.exclude=db
```

**Yaml**

```yaml
management:
  endpoint:
    health:
      group:
        custom:
          exclude: "db"
```

By default groups will inherit the same `StatusAggregator` and `HttpCodeStatusMapper` settings as the
system health, however, these can also be defined on a per-group basis. It’s also possible to 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
```

You can use `@Qualifier("groupname")` if you need to register custom `StatusAggregator` or `HttpCodeStatusMapper` beans for use with the group.

### 5.2.9. Kubernetes Probes

Applications deployed on Kubernetes can provide information about their internal state with Container Probes. Depending on your Kubernetes configuration, the kubelet will call those probes and react to the result.

Spring Boot manages your Application Availability State out-of-the-box. If deployed in a Kubernetes environment, actuator will gather the "Liveness" and "Readiness" information from the `ApplicationAvailability` interface and use that information in dedicated Health Indicators: `LivenessStateHealthIndicator` and `ReadinessStateHealthIndicator`. These indicators will be shown on the global health endpoint ("/actuator/health"). They will also be exposed as separate HTTP Probes 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: ...

<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 only enabled automatically if the application is running in a Kubernetes environment. You can enable them in any environment using the management.endpoint.health.probes.enabled configuration property.

If an application takes longer to start than the configured liveness period, Kubernetes mention the "startupProbe" as a possible solution. The "startupProbe" is not necessarily needed here as the "readinessProbe" fails until all startup tasks are done, see how Probes behave during the application lifecycle.

If your Actuator endpoints are deployed on a separate management context, be aware that endpoints are then not using 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).

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

```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 will try to solve that problem by restarting the application instance. This means that if an external system fails (e.g. a database, a Web API, an external cache), 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, i.e. Spring Boot does not include any additional health checks in the readiness probe. If the **Readiness State of an application instance** is unready, Kubernetes will not route traffic to that instance. Some external systems might not be shared by application instances, in which case they could quite naturally 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, e.g. using a circuit breaker in the caller.

If all instances of an application are unready, a Kubernetes Service with **type=ClusterIP** or **NodePort** will not accept any incoming connections. There is no HTTP error response (503 etc.) 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** will also respond in a way that depends on the implementation - the ingress service itself will have 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 is using Kubernetes **autoscaling** it may react differently to applications being taken out of the load-balancer, depending on its autoscaler configuration.

**Application lifecycle and Probes states**

An important aspect of the Kubernetes Probes support is its consistency with the application lifecycle. There is a significant different 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:

<table>
<thead>
<tr>
<th>Startup phase</th>
<th>LivenessState</th>
<th>ReadinessState</th>
<th>HTTP server</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting</td>
<td>BROKEN</td>
<td>REFUSING_TRAFFIC</td>
<td>Not started</td>
<td>Kubernetes checks the &quot;liveness&quot; Probe and restarts the application if it takes too long.</td>
</tr>
<tr>
<td>Started</td>
<td>CORRECT</td>
<td>REFUSING_TRAFFIC</td>
<td>Accepts requests</td>
<td>The application context is refreshed. The application performs startup tasks and does not receive traffic yet.</td>
</tr>
<tr>
<td>Ready</td>
<td>CORRECT</td>
<td>ACCEPTING_TRAFFIC</td>
<td>Accepts requests</td>
<td>Startup tasks are finished. The application is receiving traffic.</td>
</tr>
</tbody>
</table>

When a Spring Boot application shuts down:

<table>
<thead>
<tr>
<th>Shutdown phase</th>
<th>Liveness State</th>
<th>Readiness State</th>
<th>HTTP server</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>CORRECT</td>
<td>ACCEPTING_TRAFFIC</td>
<td>Accepts requests</td>
<td>Shutdown has been requested.</td>
</tr>
<tr>
<td>Graceful shutdown</td>
<td>CORRECT</td>
<td>REFUSING_TRAFFIC</td>
<td>New requests are rejected</td>
<td>If enabled, graceful shutdown processes in-flight requests.</td>
</tr>
<tr>
<td>Shutdown complete</td>
<td>N/A</td>
<td>N/A</td>
<td>Server is shut down</td>
<td>The application context is closed and the application is shut down.</td>
</tr>
</tbody>
</table>

Check out the Kubernetes container lifecycle section for more information about Kubernetes deployment.

5.2.10. 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 `InfoContributor` beans

The following `InfoContributor` beans are auto-configured by Spring Boot, when appropriate:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnvironmentInfoContributor</td>
<td>Exposes any key from the <code>Environment</code> under the <code>info</code> key.</td>
</tr>
<tr>
<td>GitInfoContributor</td>
<td>Exposes git information if a <code>git.properties</code> file is available.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BuildInfoContributor</td>
<td>Exposes build information if a META-INF/build-info.properties file is available.</td>
</tr>
</tbody>
</table>

It is possible to disable them all by setting the `management.info.defaults.enabled` property.

**Custom Application Information**

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**

```properties
info.app.encoding=UTF-8
info.app.java.source=11
info.app.java.target=11
```

**Yaml**

```yaml
info:
  app:
    encoding: "UTF-8"
    java:
      source: "11"
      target: "11"
```

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**

```properties
info.app.encoding=@project.build.sourceEncoding@
info.app.java.source=@java.version@
info.app.java.target=@java.version@
```

**Yaml**

```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, the info endpoint can be used to expose these properties.

A GitProperties bean is auto-configured if a git.properties file is available at the root of the classpath. See "Generate git information" for more details.

By default, the endpoint exposes git.branch, git.commit.id, and git.commit.time properties, if present. If you don't 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:

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.

The Maven and Gradle plugins can both generate that file. See "Generate build information" 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:
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 ExampleInfoContributor implements InfoContributor {
    @Override
    public void contribute(Info.Builder 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"   }
}

5.3. 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.

Actuator is supported natively with Spring MVC, Spring WebFlux, and Jersey. If both Jersey and Spring MVC are available, Spring MVC will be used.

Jackson is a required dependency in order to get the correct JSON responses as documented in the API documentation (HTML or PDF).

5.3.1. 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 shown in the following example:
The preceding application.properties example changes the endpoint from /actuator/{id} to /manage/{id} (for example, /manage/info).

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 (Servlet web applications) or spring.webflux.base-path (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"

5.3.2. 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 shown in the following example:
On Cloud Foundry, applications only receive requests on port 8080 for both HTTP and TCP routing, by default. If you want to use a custom management port on Cloud Foundry, you will need to explicitly set up the application’s routes to forward traffic to the custom port.

### 5.3.3. Configuring Management-specific SSL

When configured to use a custom port, the management server can also be configured 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 shown in the following property settings:

**Properties**

```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**

```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:
5.3.4. Customizing the Management Server Address

You can customize the address that the management endpoints are available on 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`.

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
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"
```

```properties
management.server.port=8081
management.server.address=127.0.0.1
```
5.3.5. Disabling HTTP Endpoints

If you do not want to expose endpoints over HTTP, you can set the management port to -1, as shown in the following example:

Properties

```yaml
management.server.port=-1
```

Yaml

```yaml
management:
  server:
    port: -1
```

This can be achieved using the `management.endpoints.web.exposure.exclude` property as well, as shown in the following example:

Properties

```yaml
management.endpoints.web.exposure.exclude=*  
```

Yaml

```yaml
management:
  endpoints:
    web:
      exposure:
        exclude: "*"
```

5.4. 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 and can be turned on by setting the configuration property `spring.jmx.enabled` to `true`. Spring Boot exposes management endpoints as JMX MBeans under the `org.springframework.boot` domain by default.
5.4.1. 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"
```

5.4.2. 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 shown in the following example:

**Properties**

```
management.endpoints.jmx.exposure.exclude=*
```

**Yaml**

```
management:
  endpoints:
    jmx:
      exposure:
        exclude: "*
```

5.4.3. Using Jolokia for JMX over HTTP

Jolokia is a JMX-HTTP bridge that provides an alternative method of accessing JMX beans. To use Jolokia, include a dependency to `org.jolokia:jolokia-core`. For example, with Maven, you would
add the following dependency:

```xml
<dependency>
  <groupId>org.jolokia</groupId>
  <artifactId>jolokia-core</artifactId>
</dependency>
```

The Jolokia endpoint can then be exposed by adding `jolokia` or `*` to the `management.endpoints.web.exposure.include` property. You can then access it by using `/actuator/jolokia` on your management HTTP server.

The Jolokia endpoint exposes Jolokia’s servlet as an actuator endpoint. As a result, it is specific to servlet environments such as Spring MVC and Jersey. The endpoint will not be available in a WebFlux application.

**Customizing Jolokia**

Jolokia has a number of settings that you would traditionally configure by setting servlet parameters. With Spring Boot, you can use your `application.properties` file. To do so, prefix the parameter with `management.endpoint.jolokia.config.`, as shown in the following example:

**Properties**

```properties
management.endpoint.jolokia.config.debug=true
```

**Yaml**

```yaml
management:
  endpoint:
    jolokia:
      config:
        debug: true
```

**Disabling Jolokia**

If you use Jolokia but do not want Spring Boot to configure it, set the `management.endpoint.jolokia.enabled` property to `false`, as follows:

**Properties**

```properties
management.endpoint.jolokia.enabled=false
```
Yaml

```yaml
management:
  endpoint:
    jolokia:
      enabled: false
```

5.5. 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.

5.5.1. Configure a Logger

To configure a given logger, POST a partial entity to the resource’s URI, as shown in the following example:

```
{
  "configuredLevel": "DEBUG"
}
```

To “reset” the specific level of the logger (and use the default configuration instead), you can pass a value of null as the configuredLevel.

5.6. Metrics

Spring Boot Actuator provides dependency management and auto-configuration for Micrometer, an application metrics facade that supports numerous monitoring systems, including:

- AppOptics
To learn more about Micrometer’s capabilities, please refer to its reference documentation, in particular the concepts section.

## 5.6.1. 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. For example, to disable Datadog:

```
Properties

management.metrics.export.datadog.enabled=false
```
You can also disable all registries unless stated otherwise by the registry-specific property, as shown in the following example:

```
Properties

management.metrics.export.defaults.enabled=false
```

```
Yaml

management:
  metrics:
    export:
      datadog:
        enabled: false
```

Spring Boot will also add 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:

```
@Bean
MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {
    return registry -> registry.config().commonTags("region", "us-east-1");
}
```

You can apply customizations to particular registry implementations by being more specific about the generic type:
Spring Boot also configures built-in instrumentation that you can control via configuration or dedicated annotation markers.

### 5.6.2. Supported monitoring systems

**AppOptics**

By default, the AppOptics registry pushes metrics to `api.appoptics.com/v1/measurements` periodically. To export metrics to SaaS AppOptics, your API token must be provided:

**Properties**

```
management.metrics.export.appoptics.api-token=YOUR_TOKEN
```

**Yaml**

```
management:
  metrics:
    export:
      appoptics:
        api-token: "YOUR_TOKEN"
```

**Atlas**

By default, metrics are exported to Atlas running on your local machine. The location of the Atlas server to use can be provided using:

**Properties**

```
management.metrics.export.atlas.uri=https://atlas.example.com:7101/api/v1/publish
```

**Yaml**

```
management:
  metrics:
    export:
      atlas:
        uri: "https://atlas.example.com:7101/api/v1/publish"
```

**Datadog**

Datadog registry pushes metrics to `datadoghq` periodically. To export metrics to Datadog, your API
key must be provided:

**Properties**

```
management.metrics.export.datadog.api-key=YOUR_KEY
```

**Yaml**

```
management:
  metrics:
    export:
      datadog:
        api-key: "YOUR_KEY"
```

You can also change the interval at which metrics are sent to Datadog:

**Properties**

```
management.metrics.export.datadog.step=30s
```

**Yaml**

```
management:
  metrics:
    export:
      datadog:
        step: "30s"
```

**Dynatrace**

Dynatrace registry pushes metrics to the configured URI periodically. To export metrics to Dynatrace, your API token, device ID, and URI must be provided:

**Properties**

```
management.metrics.export.dynatrace.api-token=YOUR_TOKEN
management.metrics.export.dynatrace.device-id=YOUR_DEVICE_ID
management.metrics.export.dynatrace.uri=YOUR_URI
```
You can also change the interval at which metrics are sent to Dynatrace:

**Properties**

```
management.metrics.export.dynatrace.step=30s
```

**Yaml**

```yaml
management:
  metrics:
    export:
      dynatrace:
        step: "30s"
```

**Elastic**

By default, metrics are exported to Elastic running on your local machine. The location of the Elastic server to use can be provided using the following property:

**Properties**

```
management.metrics.export.elastic.host=https://elastic.example.com:8086
```

**Yaml**

```yaml
management:
  metrics:
    export:
      elastic:
        host: "https://elastic.example.com:8086"
```

**Ganglia**

By default, metrics are exported to Ganglia running on your local machine. The Ganglia server host and port to use can be provided using:
Graphite

By default, metrics are exported to Graphite running on your local machine. The Graphite server host and port to use can be provided using:

Properties

management.metrics.export.graphite.host=graphite.example.com
management.metrics.export.graphite.port=9004

Yaml

management:
  metrics:
    export:
      graphite:
        host: "graphite.example.com"
        port: 9004

Micrometer provides a default HierarchicalNameMapper that governs how a dimensional meter id is mapped to flat hierarchical names.

To take control over this behaviour, define your GraphiteMeterRegistry and supply your own HierarchicalNameMapper. An auto-configured GraphiteConfig and Clock beans are provided unless you define your own:

@Bean
public GraphiteMeterRegistry graphiteMeterRegistry(GraphiteConfig config, Clock clock) {
  return new GraphiteMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);
}
Humio

By default, the Humio registry pushes metrics to cloud.humio.com periodically. To export metrics to SaaS Humio, your API token must be provided:

**Properties**

```
management.metrics.export.humio.api-token=YOUR_TOKEN
```

**Yaml**

```
management:
  metrics:
    export:
      humio:
        api-token: "YOUR_TOKEN"
```

You should also configure one or more tags to identify the data source to which metrics will be pushed:

**Properties**

```
management.metrics.export.humio.tags.alpha=a
management.metrics.export.humio.tags.bravo=b
```

**Yaml**

```
management:
  metrics:
    export:
      humio:
        tags:
          alpha: "a"
          bravo: "b"
```

Influx

By default, metrics are exported to Influx running on your local machine. The location of the Influx server to use can be provided using:

**Properties**

```
management.metrics.export.influx.uri=https://influx.example.com:8086
```
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. The domain to use can be provided using:

**Properties**

```java
management.metrics.export.jmx.domain=com.example.app.metrics
```

Micrometer provides a default `HierarchicalNameMapper` that governs how a dimensional meter id is mapped to flat hierarchical names.

To take control over this behaviour, define your `JmxMeterRegistry` and supply your own `HierarchicalNameMapper`. An auto-configured `JmxConfig` and `Clock` beans are provided unless you define your own:

```java
@Bean
public JmxMeterRegistry jmxMeterRegistry(JmxConfig config, Clock clock) {
    return new JmxMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);
}
```

**KairosDB**

By default, metrics are exported to KairosDB running on your local machine. The location of the KairosDB server to use can be provided using:
**Properties**

management.metrics.export.kairos.uri=https://kairosdb.example.com:8080/api/v1/datapoints

**Yaml**

```yaml
management:
  metrics:
    export:
      kairos:
        uri: "https://kairosdb.example.com:8080/api/v1/datapoints"
```

**New Relic**

New Relic registry pushes metrics to New Relic periodically. To export metrics to New Relic, your API key and account id must be provided:

**Properties**

management.metrics.export.newrelic.api-key=YOUR_KEY
management.metrics.export.newrelic.account-id=YOUR_ACCOUNT_ID

**Yaml**

```yaml
management:
  metrics:
    export:
      newrelic:
        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.metrics.export.newrelic.step=30s

**Yaml**

```yaml
management:
  metrics:
    export:
      newrelic:
        step: "30s"
```

By default, metrics are published via REST calls but it is also possible to use the Java Agent API if
you have it on the classpath:

Properties

management.metrics.export.newrelic.client-provider-type=insights-agent

Yaml

```yaml
management:
  metrics:
    export:
      newrelic:
        client-provider-type: "insights-agent"
```

Finally, you can take full control by defining your own `NewRelicClientProvider` bean.

Prometheus

Prometheus expects to scrape or poll individual app instances for metrics. Spring Boot provides an actuator endpoint available at `/actuator/prometheus` to present a Prometheus scrape with the appropriate format.

The endpoint is not available by default and must be exposed, see exposing endpoints for more details.

Here is an example `scrape_config` to add to `prometheus.yml`:

```yaml
scrape_configs:
  - job_name: 'spring'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['HOST:PORT']
```

For ephemeral or batch jobs which may not exist long enough to be scraped, Prometheus Pushgateway support can be used to expose their metrics to Prometheus. To enable Prometheus Pushgateway support, add the following dependency to your project:

```xml
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient_pushgateway</artifactId>
</dependency>
```

When the Prometheus Pushgateway dependency is present on the classpath and the `management.metrics.export.prometheus.pushgateway.enabled` property is set to `true`, a `PrometheusPushGatewayManager` bean is auto-configured. This manages the pushing of metrics to a Prometheus Pushgateway.
The PrometheusPushGatewayManager can be tuned using properties under management.metrics.export.prometheus.pushgateway. For advanced configuration, you can also provide your own PrometheusPushGatewayManager bean.

**SignalFx**

SignalFx registry pushes metrics to SignalFx periodically. To export metrics to SignalFx, your access token must be provided:

**Properties**

```
management.metrics.export.signalfx.access-token=YOUR_ACCESS_TOKEN
```

**Yaml**

```
management:
  metrics:
    export:
      signalfx:
        access-token: "YOUR_ACCESS_TOKEN"
```

You can also change the interval at which metrics are sent to SignalFx:

**Properties**

```
management.metrics.export.signalfx.step=30s
```

**Yaml**

```
management:
  metrics:
    export:
      signalfx:
        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 allows you to see what metrics are collected in the metrics endpoint.

The in-memory backend disables itself as soon as you’re using any of the other available backend. You can also disable it explicitly:

**Properties**

```
management.metrics.export.simple.enabled=false
```
Yaml

```yaml
management:
  metrics:
    export:
      simple:
        enabled: false
```

**Stackdriver**

Stackdriver registry pushes metrics to Stackdriver periodically. To export metrics to SaaS Stackdriver, your Google Cloud project id must be provided:

**Properties**

```bash
management.metrics.export.stackdriver.project-id=my-project
```

Yaml

```yaml
management:
  metrics:
    export:
      stackdriver:
        project-id: "my-project"
```

You can also change the interval at which metrics are sent to Stackdriver:

**Properties**

```bash
management.metrics.export.stackdriver.step=30s
```

Yaml

```yaml
management:
  metrics:
    export:
      stackdriver:
        step: "30s"
```

**StatsD**

The StatsD registry pushes metrics over UDP to a StatsD agent eagerly. By default, metrics are exported to a StatsD agent running on your local machine. The StatsD agent host, port, and protocol to use can be provided using:
Properties

management.metrics.export.statsd.host=statsd.example.com
management.metrics.export.statsd.port=9125
management.metrics.export.statsd.protocol=udp

Yaml

management:
  metrics:
    export:
      statsd:
        host: "statsd.example.com"
        port: 9125
        protocol: "udp"

You can also change the StatsD line protocol to use (default to Datadog):

Properties

management.metrics.export.statsd.flavor=etsy

Yaml

management:
  metrics:
    export:
      statsd:
        flavor: "etsy"

Wavefront

Wavefront registry pushes metrics to Wavefront periodically. If you are exporting metrics to Wavefront directly, your API token must be provided:

Properties

management.metrics.export.wavefront.api-token=YOUR_API_TOKEN

Yaml

management:
  metrics:
    export:
      wavefront:
        api-token: "YOUR_API_TOKEN"
Alternatively, you may use a Wavefront sidecar or an internal proxy set up in your environment that forwards metrics data to the Wavefront API host:

**Properties**

```
management.metrics.export.wavefront.uri=proxy://localhost:2878
```

**Yaml**

```
management:
  metrics:
    export:
      wavefront:
        uri: "proxy://localhost:2878"
```

If publishing metrics to a Wavefront proxy (as described in the 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.metrics.export.wavefront.step=30s
```

**Yaml**

```
management:
  metrics:
    export:
      wavefront:
        step: "30s"
```

### 5.6.3. Supported Metrics

Spring Boot registers the following core metrics when applicable:

- JVM metrics, report utilization of:
  - Various memory and buffer pools
  - Statistics related to garbage collection
  - Threads utilization
  - Number of classes loaded/unloaded
- CPU metrics
- File descriptor metrics
- Kafka consumer and producer metrics
• Log4j2 metrics: record the number of events logged to Log4j2 at each level
• Logback metrics: record the number of events logged to Logback at each level
• Uptime metrics: report a gauge for uptime and a fixed gauge representing the application’s absolute start time
• Tomcat metrics (server.tomcat.mbeanregistry.enabled must be set to true for all Tomcat metrics to be registered)
• Spring Integration metrics

**Spring MVC Metrics**

Auto-configuration enables the instrumentation of requests handled by Spring MVC. When management.metrics.web.server.request.autotime.enabled is true, this instrumentation occurs for all requests. Alternatively, when set to false, you can enable instrumentation by adding @Timed to a request-handling method:

```java
@RestController
@Timed ➊
public class MyController {

    @GetMapping("/api/people")
    @Timed(extraTags = { "region", "us-east-1" }) ➋
    @Timed(value = "all.people", longTask = true) ➌
    public List<Person> listPeople() { ... }
}
```

➊ A controller class to enable timings on every request handler in the controller.

➋ A method to enable for an individual endpoint. This is not necessary if you have it on the class, but can be used to further customize the timer for this particular endpoint.

➌ A method with longTask = true to enable 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, metrics are generated with the name, http.server.requests. The name can be customized by setting the management.metrics.web.server.request.metric-name property.

By default, Spring MVC-related metrics are tagged with the following information:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception</td>
<td>Simple class name of any exception that was thrown while handling the request.</td>
</tr>
<tr>
<td>method</td>
<td>Request’s method (for example, GET or POST)</td>
</tr>
<tr>
<td>outcome</td>
<td>Request’s outcome based on the status code of the response. 1xx is INFORMATIONAL, 2xx is SUCCESS, 3xx is REDIRECTION, 4xx CLIENT_ERROR, and 5xx is SERVER_ERROR</td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>status</td>
<td>Response’s HTTP status code (for example, 200 or 500)</td>
</tr>
<tr>
<td>uri</td>
<td>Request’s URI template prior to variable substitution, if possible (for example, /api/person/{id})</td>
</tr>
</tbody>
</table>

To add to the default tags, provide one or more `@Bean`s that implement `WebMvcTagsContributor`. To replace the default tags, provide a `@Bean` that implements `WebMvcTagsProvider`.

**Spring WebFlux Metrics**

Auto-configuration enables the instrumentation of all requests handled by 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.metrics.web.server.request.metric-name` property.

By default, WebFlux-related metrics are tagged with the following information:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception</td>
<td>Simple class name of any exception that was thrown while handling the request.</td>
</tr>
<tr>
<td>method</td>
<td>Request's method (for example, GET or POST)</td>
</tr>
<tr>
<td>outcome</td>
<td>Request's outcome based on the status code of the response. 1xx is INFORMATIONAL, 2xx is SUCCESS, 3xx is REDIRECTION, 4xx CLIENT_ERROR, and 5xx is SERVER_ERROR</td>
</tr>
<tr>
<td>status</td>
<td>Response’s HTTP status code (for example, 200 or 500)</td>
</tr>
<tr>
<td>uri</td>
<td>Request’s URI template prior to variable substitution, if possible (for example, /api/person/{id})</td>
</tr>
</tbody>
</table>

To add to the default tags, provide one or more `@Bean`s that implement `WebFluxTagsContributor`. To replace the default tags, provide a `@Bean` that implements `WebFluxTagsProvider`.

**Jersey Server Metrics**

When Micrometer's `micrometer-jersey2` module is on the classpath, auto-configuration enables the instrumentation of requests handled by the Jersey JAX-RS implementation. When `management.metrics.web.server.request.autotime.enabled` is `true`, this instrumentation occurs for all requests. Alternatively, when set to `false`, you can enable instrumentation by adding `@Timed` to a request-handling method:
@Component
@Path("/api/people")
@Timed
public class Endpoint {
    @GET
    @Timed(extraTags = { "region", "us-east-1" }) ②
    @Timed(value = "all.people", longTask = true) ③
    public List<Person> listPeople() { ... }
}

① On a resource class to enable timings on every request handler in the resource.
② On a method to enable for an individual endpoint. This is not necessary if you have it on the class, but can be used to further customize the timer for this particular endpoint.
③ On a method with `longTask = true` to enable 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, metrics are generated with the name, `http.server.requests`. The name can be customized by setting the `management.metrics.web.server.request.metric-name` property.

By default, Jersey server metrics are tagged with the following information:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception</td>
<td>Simple class name of any exception that was thrown while handling the request.</td>
</tr>
<tr>
<td>method</td>
<td>Request's method (for example, GET or POST)</td>
</tr>
<tr>
<td>outcome</td>
<td>Request's outcome based on the status code of the response. 1xx is INFORMATIONAL, 2xx is SUCCESS, 3xx is REDIRECTION, 4xx CLIENT_ERROR, and 5xx is SERVER_ERROR</td>
</tr>
<tr>
<td>status</td>
<td>Response's HTTP status code (for example, 200 or 500)</td>
</tr>
<tr>
<td>uri</td>
<td>Request's URI template prior to variable substitution, if possible (for example, /api/person/{id})</td>
</tr>
</tbody>
</table>

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 get injected with an auto-configured builder and use it to create instances:

- `RestTemplateBuilder` for `RestTemplate`

It is also possible to apply manually the customizers responsible for this instrumentation, namely
MetricsRestTemplateCustomizer and MetricsWebClientCustomizer.

By default, metrics are generated with the name, http.client.requests. The name can be customized by setting the management.metrics.web.client.request.metric-name property.

By default, metrics generated by an instrumented client are tagged with the following information:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clientName</td>
<td>Host portion of the URI</td>
</tr>
<tr>
<td>method</td>
<td>Request's method (for example, GET or POST)</td>
</tr>
<tr>
<td>outcome</td>
<td>Request's outcome based on the status code of the response. 1xx is INFORMATIONAL, 2xx is SUCCESS, 3xx is REDIRECTION, 4xx CLIENT_ERROR, and 5xx is SERVER_ERROR, UNKNOWN otherwise</td>
</tr>
<tr>
<td>status</td>
<td>Response's HTTP status code if available (for example, 200 or 500), or IO_ERROR in case of I/O issues, CLIENT_ERROR otherwise</td>
</tr>
<tr>
<td>uri</td>
<td>Request's URI template prior to variable substitution, if possible (for example, /api/person/{id})</td>
</tr>
</tbody>
</table>

To customize the tags, and depending on your choice of client, you can provide a @Bean that implements RestTemplateExchangeTagsProvider or WebClientExchangeTagsProvider. There are convenience static functions in RestTemplateExchangeTags and WebClientExchangeTags.

**Cache Metrics**

Auto-configuration enables the instrumentation of all available Caches 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:

- Caffeine
- EhCache 2
- Hazelcast
- Any compliant JCache (JSR-107) implementation
- Redis

Metrics are tagged by the name of the cache and by the name of the CacheManager that is derived from the bean name.

Only caches that are configured on startup are bound to the registry. For caches not defined in the cache’s configuration, e.g. 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.
**DataSource Metrics**

Auto-configuration enables the instrumentation of all available `DataSource` objects with metrics prefixed with `jdbc.connections`. Data source instrumentation results in gauges representing 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.

By default, Spring Boot provides metadata for all supported data sources; you can add additional `DataSourcePoolMetadataProvider` beans if your favorite data source isn’t supported out of the box. See `DataSourcePoolMetadataProvidersConfiguration` for examples.

Also, Hikari-specific metrics are exposed with a `hikaricp` prefix. Each metric is tagged by the name of the Pool (can be controlled with `spring.datasource.name`).

**Hibernate Metrics**

Auto-configuration enables the instrumentation of all available Hibernate `EntityManagerFactory` instances that have statistics enabled with a metric named `hibernate`.

Metrics are also tagged by the name of the `EntityManagerFactory` that 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` as shown in the following example:

**Properties**

```properties
spring.jpa.properties[hibernate.generate_statistics]=true
```

**Yaml**

```yaml
spring:
  jpa:
    properties:
      "[hibernate.generate_statistics]": true
```

**RabbitMQ Metrics**

Auto-configuration will enable the instrumentation of all available RabbitMQ connection factories with a metric named `rabbitmq`.

**Kafka Metrics**

Auto-configuration will register a `MicrometerConsumerListener` and `MicrometerProducerListener` for the auto-configured consumer factory and producer factory respectively. For more details refer to `Micrometer Native Metrics` section of the Spring Kafka documentation.
5.6.4. Registering custom metrics

To register custom metrics, inject `MeterRegistry` into your component, as shown in the following example:

```java
class Dictionary {
    private final List<String> words = new CopyOnWriteArrayList<>();

    Dictionary(MeterRegistry registry) {
        registry.gaugeCollectionSize("dictionary.size", Tags.empty(), this.words);
    }

    // ...
}
```

If your metrics depend on other beans, it is recommended that you use a `MeterBinder` to register them, as shown in the following example:

```java
@Bean
MeterBinder queueSize(Queue queue) {
    return (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. By default, metrics from all `MeterBinder` beans will be automatically bound to the Spring-managed `MeterRegistry`. A `MeterBinder` implementation can also be useful if you find that you repeatedly instrument a suite of metrics across components or applications.

5.6.5. Customizing individual metrics

If you need to apply customizations to specific `Meter` instances you can use the `io.micrometer.core.instrument.config.MeterFilter` interface. By default, all `MeterFilter` beans will be automatically applied to the micrometer `MeterRegistry.Config`.

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
@Bean
public MeterFilter renameRegionTagMeterFilter() {
    return MeterFilter.renameTag("com.example", "mytag.region", "mytag.area");
}
```
Common tags

Common tags are generally used for dimensional drill-down on the operating environment like host, instance, region, stack, etc. Commons tags are applied to all meters and can be configured as shown in the following example:

Properties

```
management.metrics.tags.region=us-east-1
management.metrics.tags.stack=prod
```

Yaml

```
management:
  metrics:
    tags:
      region: "us-east-1"
      stack: "prod"
```

The example above adds region and stack tags to all meters with a value of us-east-1 and prod respectively.

The order of common tags is important if you are using Graphite. As the order of common tags cannot be guaranteed using this approach, Graphite users are advised to define a custom MeterFilter instead.

Per-meter properties

In addition to MeterFilter beans, it's also possible to apply a limited set of customization on a per-meter basis using properties. Per-meter customizations apply to any all meter IDs that start with the given name. For example, the following will disable 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>management.metrics.enable</td>
<td>Whether to deny meters from emitting any metrics.</td>
</tr>
<tr>
<td>management.metrics.distribution.percentiles-histogram</td>
<td>Whether to publish a histogram suitable for computing aggregable (across dimension) percentile approximations.</td>
</tr>
<tr>
<td>management.metrics.distribution.minimum-expected-value,</td>
<td>Publish less histogram buckets by clamping the range of expected values.</td>
</tr>
<tr>
<td>management.metrics.distribution.maximum-expected-value</td>
<td></td>
</tr>
<tr>
<td>management.metrics.distribution.percentiles</td>
<td>Publish percentile values computed in your application</td>
</tr>
<tr>
<td>management.metrics.distribution.slo</td>
<td>Publish a cumulative histogram with buckets defined by your service-level objectives.</td>
</tr>
</tbody>
</table>

For more details on concepts behind percentiles-histogram, percentiles and slo refer to the "Histograms and percentiles" section of the micrometer documentation.

### 5.6.6. Metrics endpoint

Spring Boot provides a metrics endpoint that can be used 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, e.g. `/actuator/metrics/jvm.memory.max`.

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 it is shipped to. 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, e.g. `/actuator/metrics/jvm.memory.max?tag=area:nonheap`.

The reported measurements are the sum of the statistics of all meters matching the meter name and any tags that have been applied. So in the example above, the returned "Value" statistic is the sum of the maximum memory footprints of "Code Cache", "Compressed Class Space", and "Metaspace" areas of the heap. If you only wanted to see the maximum size for the "Metaspace", you could add an additional `tag=id:Metaspace`, i.e. `/actuator/metrics/jvm.memory.max?tag=area:nonheap&tag=id:Metaspace`. 
5.7. 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.

Auditing can be enabled 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.

5.7.1. 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).

5.8. HTTP Tracing

HTTP Tracing can be enabled by providing a bean of type HttpTraceRepository in your application’s configuration. For convenience, Spring Boot offers an InMemoryHttpTraceRepository that stores traces for the last 100 request-response exchanges, by default. InMemoryHttpTraceRepository is limited compared to other tracing solutions and we recommend using it only for development environments. For production environments, use of a production-ready tracing or observability solution, such as Zipkin or Spring Cloud Sleuth, is recommended. Alternatively, create your own HttpTraceRepository that meets your needs.

The httptrace endpoint can be used to obtain information about the request-response exchanges that are stored in the HttpTraceRepository.

5.8.1. Custom HTTP tracing

To customize the items that are included in each trace, use the management.trace.http.include configuration property. For advanced customization, consider registering your own HttpExchangeTracer implementation.

5.9. 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 containing the application PID (by default, in the
application directory with a file name of \texttt{application.pid}).

- \texttt{WebServerPortFileWriter} creates a file (or files) containing the ports of the running web server (by default, in the application directory with a file name of \texttt{application.port}).

By default, these writers are not activated, but you can enable:

- \texttt{By Extending Configuration}
- \texttt{Programmatically}

### 5.9.1. Extending Configuration

In the \texttt{META-INF/spring.factories} file, you can activate the listener(s) that writes a PID file, as shown in the following example:

\begin{verbatim}
org.springframework.context.ApplicationListener=\org.springframework.boot.context.ApplicationPidFileWriter,\
org.springframework.boot.web.context.WebServerPortFileWriter
\end{verbatim}

### 5.9.2. Programmatically

You can also activate a listener by invoking the \texttt{SpringApplication.addListeners(…)} method and passing the appropriate \texttt{Writer} object. This method also lets you customize the file name and path in the \texttt{Writer} constructor.

### 5.10. Cloud Foundry Support

Spring Boot’s actuator module includes additional support that is activated when you deploy to a compatible Cloud Foundry instance. The \texttt{/cloudfoundryapplication} path provides an alternative secured route to all \texttt{@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 may include full health information instead of the typical “running” or “stopped” status.

The \texttt{/cloudfoundryapplication} path is not directly accessible to regular users. In order to use the endpoint, a valid UAA token must be passed with the request.

### 5.10.1. Disabling Extended Cloud Foundry Actuator Support

If you want to fully disable the \texttt{/cloudfoundryapplication} endpoints, you can add the following setting to your \texttt{application.properties} file:

\begin{verbatim}
application.properties (Properties)

management.cloudfoundry.enabled=false
\end{verbatim}
5.10.2. 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
```

5.10.3. Custom context path

If the server's context-path has been configured to anything other than `/`, the Cloud Foundry endpoints will not be available at the root of the application. For example, if `server.servlet.context-path=/app`, Cloud Foundry endpoints will be 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 will need to explicitly configure that in your application. The configuration will differ depending on the web server in use. For Tomcat, the following configuration can be added:
@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 (c, 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("/*");
    };
}

5.11. 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 6. 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 container images (such as Docker), or to virtual/real machines.

This section covers some of the more common deployment scenarios.

6.1. Deploying to Containers

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. At runtime you shouldn’t 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
```

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.

More efficient container images can also be created by creating separate layers for your dependencies and application classes and resources (which normally change more frequently).

6.2. 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.

### 6.2.1. 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:

```bash
$ cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

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)
----- Downloading Open Jdk JRE 1.8.0_121 from https://java-buildpack.cloudfoundry.org/openjdk/trusty/x86_64/openjdk-1.8.0_121.tar.gz (found in cache)
  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

<table>
<thead>
<tr>
<th>name</th>
<th>requested state</th>
<th>instances</th>
<th>memory</th>
<th>disk</th>
<th>urls</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>acloudyspringtime</td>
<td>started</td>
<td>1/1</td>
<td>512M</td>
<td>1G</td>
<td>acloudyspringtime.cfapps.io</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
@Component
class MyBean implements EnvironmentAware {

    private String instanceId;

    @Override
    public void setEnvironment(Environment environment) {
        this.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.

The [Java CFEnv](https://github.com/freemix/JavaCFEnv) project is a better fit for tasks such as configuring a DataSource.

### 6.2.2. 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlaxon.html#end2end.kubernetes).

**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 via the PodSpec in the pod's configuration file as follows:

```yaml
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.

### 6.2.3. 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:

```shell
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 master

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 1.8... done
-----> Installing Maven 3.3.1... 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] 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]  master -> master

Your application should now be up and running on Heroku. For more details, refer to Deploying Spring Boot Applications to Heroku.

6.2.4. OpenShift

OpenShift has many resources describing how to deploy Spring Boot applications, including:

- Using the S2I builder
- Architecture guide
6.2.5. 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 the 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
```

**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:

```yaml
deploy:
  artifact: target/demo-0.0.1-SNAPSHOT.jar
```
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.

6.2.6. Boxfuse and Amazon Web Services

Boxfuse 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. Boxfuse 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. Boxfuse 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 Boxfuse account, connected it to your AWS account, installed the latest version of the Boxfuse 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.

By default, Boxfuse activates a Spring profile named boxfuse on startup. If your executable jar or war contains an application-boxfuse.properties file, Boxfuse bases its configuration on the properties it contains.

At this point, boxfuse 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:
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 Boxfuse Spring Boot integration to get started with a Maven build to run the app.

### 6.2.7. 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 run in App Engine, you can create a project in the UI first, which sets up a unique identifier for you and also sets up HTTP routes. Add a Java app to the project and leave it empty and then use the Google Cloud SDK to push your Spring Boot app into that slot from the command line or CI build.

App Engine Standard requires you to use WAR packaging. Follow these steps to deploy App Engine Standard application to Google Cloud.

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:
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:

```xml
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>appengine-maven-plugin</artifactId>
    <version>1.3.0</version>
    <configuration>
        <project>myproject</project>
    </configuration>
</plugin>
```

Then deploy with `mvn appengine:deploy` (if you need to authenticate first, the build fails).

### 6.3. Installing Spring Boot Applications

In addition to running Spring Boot applications by using `java -jar`, it is also possible to make fully executable applications for Unix systems. A fully executable jar can be executed like any other executable binary or it can be registered with `init.d` or `systemd`. This helps when installing and managing Spring Boot applications in common production environments.
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.

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:

```xml
<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:

```groovy
bootJar {
    launchScript()
}
```

You can then run your application by typing `./my-application.jar` (where `my-application` is the name of your artifact). The directory containing the jar is used as your application's working directory.

### 6.3.1. Supported Operating Systems

The default script 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 `embeddedLaunchScript`.

### 6.3.2. Unix/Linux Services

Spring Boot application can be easily started as Unix/Linux services by using either `init.d` or `systemd`.

**Installation as an init.d Service (System V)**

If you configured Spring Boot's Maven or Gradle plugin to generate a fully executable jar, and you do not use a custom `embeddedLaunchScript`, your application can be used as an `init.d` service. To do
so, symlink the jar to init.d to support the standard start, stop, restart, and status commands.

The script 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

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

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.
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:

```bash
$ chmod 500 your-app.jar
```

Second, you should also take steps to limit the damage if your application or the account that's 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:

```bash
$ 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:

```bash
$ chmod 400 your-app.conf
$ sudo chown root:root your-app.conf
```

**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. Although you can continue to use `init.d` scripts with `systemd`, it is also possible to launch Spring Boot applications by using `systemd` 'service' scripts.

Assuming that you have a Spring Boot application installed in `/var/myapp`, to install a Spring Boot application 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

[Service]
User=myapp
ExecStart=/var/myapp/myapp.jar
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target

Remember to change the **Description**, **User**, and **ExecStart** fields for your application.

The **ExecStart** field does not declare the script action command, which means that the **run** command is used by default.

Note that, unlike when running as an **init.d** service, 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
```

Refer to **man systemctl** for more details.

**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:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Gradle default</th>
<th>Maven default</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mode</strong></td>
<td>The script mode.</td>
<td>auto</td>
<td>auto</td>
</tr>
<tr>
<td><strong>initInfo</strong></td>
<td>Provides the <code>Provides</code> section of “INIT INFO”</td>
<td><code>${task.baseName}</code></td>
<td><code>${project.artifactId}</code></td>
</tr>
<tr>
<td><strong>initInfo</strong></td>
<td>Required-Start section of “INIT INFO”.</td>
<td>$remote_fs $syslog $network</td>
<td>$remote_fs $syslog $network</td>
</tr>
<tr>
<td><strong>initInfo</strong></td>
<td>Required-Stop section of “INIT INFO”.</td>
<td>$remote_fs $syslog $network</td>
<td>$remote_fs $syslog $network</td>
</tr>
<tr>
<td><strong>initInfo</strong></td>
<td>Default-Start section of “INIT INFO”.</td>
<td>2 3 4 5</td>
<td>2 3 4 5</td>
</tr>
<tr>
<td><strong>initInfo</strong></td>
<td>Default-Stop section of “INIT INFO”.</td>
<td>0 1 6</td>
<td>0 1 6</td>
</tr>
<tr>
<td><strong>initInfo</strong></td>
<td>Short-Description section of “INIT INFO”.</td>
<td>Single-line version of <code>project.description</code> (falling back to <code>${task.baseName}</code>)</td>
<td><code>${project.name}</code></td>
</tr>
<tr>
<td><strong>initInfo</strong></td>
<td>Description section of “INIT INFO”.</td>
<td><code>${project.description}</code> (falling back to <code>${task.baseName}</code>)</td>
<td><code>${project.description}</code> (falling back to <code>${project.name}</code>)</td>
</tr>
<tr>
<td><strong>chkconfig</strong></td>
<td>Reference to a file script that should be inlined in the default launch script. This can be used to set environmental variables such as <code>JAVA_OPTS</code> before any external config files are loaded</td>
<td>2345 99 01</td>
<td>2345 99 01</td>
</tr>
<tr>
<td><strong>confFolder</strong></td>
<td>The default value for <code>CONF_FOLDER</code></td>
<td>Folder containing the jar</td>
<td>Folder containing the jar</td>
</tr>
<tr>
<td><strong>inlinedConfigScript</strong></td>
<td>Reference to a file script that should be inlined in the default launch script. This can be used to set environmental variables such as <code>JAVA_OPTS</code> before any external config files are loaded</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>logFolder</strong></td>
<td>Default value for <code>LOG_FOLDER</code>. Only valid for an <code>init.d</code> service</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>logFileName</strong></td>
<td>Default value for <code>LOG_FILENAME</code>. Only valid for an <code>init.d</code> service</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pidFolder</strong></td>
<td>Default value for <code>PID_FOLDER</code>. Only valid for an <code>init.d</code> service</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Gradle default</td>
<td>Maven default</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------</td>
<td>---------------</td>
</tr>
<tr>
<td>pidFilename</td>
<td>Default value for the name of the PID file in PID_FOLDER. Only valid for an init.d service</td>
<td></td>
<td></td>
</tr>
<tr>
<td>useStartStopDaemon</td>
<td>Whether the start-stop-daemon command, when it's available, should be used to control the process</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>stopWaitTime</td>
<td>Default value for STOP_WAIT_TIME in seconds. Only valid for an init.d service</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODE</td>
<td>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</td>
</tr>
<tr>
<td>RUN_AS_USER</td>
<td>The user that will be used to run the application. When not set, the user that owns the jar file will be used.</td>
</tr>
<tr>
<td>USE_START_STOP_DAEMON</td>
<td>Whether the start-stop-daemon command, when it's available, should be used to control the process. Defaults to true.</td>
</tr>
<tr>
<td>PID_FOLDER</td>
<td>The root name of the pid folder (/var/run by default).</td>
</tr>
<tr>
<td>LOG_FOLDER</td>
<td>The name of the folder in which to put log files (/var/log by default).</td>
</tr>
<tr>
<td>CONF_FOLDER</td>
<td>The name of the folder from which to read .conf files (same folder as jar-file by default).</td>
</tr>
<tr>
<td>LOG_FILENAME</td>
<td>The name of the log file in the LOG_FOLDER (&lt;appname&gt;.log by default).</td>
</tr>
<tr>
<td>APP_NAME</td>
<td>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.</td>
</tr>
<tr>
<td>RUN_ARGS</td>
<td>The arguments to pass to the program (the Spring Boot app).</td>
</tr>
<tr>
<td>JAVA_HOME</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Variable Description

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA_OPTS</td>
<td>Options that are passed to the JVM when it is launched.</td>
</tr>
<tr>
<td>JARFILE</td>
<td>The explicit location of the jar file, in case the script is being used to launch a jar that it is not actually embedded.</td>
</tr>
<tr>
<td>DEBUG</td>
<td>If not empty, sets the <code>-x</code> flag on the shell process, allowing you to see the logic in the script.</td>
</tr>
<tr>
<td>STOP_WAIT_TIME</td>
<td>The time in seconds to wait when stopping the application before forcing a shutdown (60 by default).</td>
</tr>
</tbody>
</table>

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.

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
```

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](#).

### 6.3.3. 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.

### 6.4. What to Read Next

Check out 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 [Spring Boot CLI](#), or you can jump ahead to read about [build tool plugins](#).
Chapter 7. Spring Boot CLI

The Spring Boot CLI is a command line tool that you can use if you want to quickly develop a Spring application. It lets you run Groovy scripts, which means that you have a familiar Java-like syntax without so much boilerplate code. You can also bootstrap a new project or write your own command for it.

7.1. 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.

7.2. 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:

run [options] <files> [--] [args]
    Run a spring groovy script

... more command help is shown here
```

You can type `spring help` to get more details about any of the supported commands, as shown in the following example:
$ spring help run
spring run - Run a spring groovy script

usage: spring run [options] <files> [-] [args]

Option                     Description
------                     -----------
--autoconfigure [Boolean]  Add autoconfigure compiler
                            transformations (default: true)
--classpath, -cp           Additional classpath entries
--no-guess-dependencies   Do not attempt to guess dependencies
--no-guess-imports        Do not attempt to guess imports
-q, --quiet               Quiet logging
-v, --verbose             Verbose logging of dependency
                            resolution
--watch                   Watch the specified file for changes

The version command provides a quick way to check which version of Spring Boot you are using, as follows:

$ spring version
Spring CLI v2.4.2

7.2.1. Running Applications with the CLI

You can compile and run Groovy source code by using the run command. The Spring Boot CLI is
completely self-contained, so you do not need any external Groovy installation.

The following example shows a “hello world” web application written in Groovy:

hello.groovy

```
@RestController
class WebApplication {

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

}
```

To compile and run the application, type the following command:

$ spring run hello.groovy
To pass command-line arguments to the application, use `--` to separate the commands from the “spring” command arguments, as shown in the following example:

```
$ spring run hello.groovy -- --server.port=9000
```

To set JVM command line arguments, you can use the `JAVA_OPTS` environment variable, as shown in the following example:

```
$ JAVA_OPTS=-Xmx1024m spring run hello.groovy
```

When setting `JAVA_OPTS` on Microsoft Windows, make sure to quote the entire instruction, such as `set "JAVA_OPTS=-Xms256m -Xmx2048m"`. Doing so ensures the values are properly passed to the process.

**Deduced “grab” Dependencies**

Standard Groovy includes a `@Grab` annotation, which lets you declare dependencies on third-party libraries. This useful technique lets Groovy download jars in the same way as Maven or Gradle would without requiring you to use a build tool.

Spring Boot extends this technique further and tries to deduce which libraries to “grab” based on your code. For example, since the `WebApplication` code shown previously uses `@RestController` annotations, Spring Boot grabs "Tomcat" and "Spring MVC".

The following items are used as “grab hints”:

<table>
<thead>
<tr>
<th>Items</th>
<th>Grabs</th>
</tr>
</thead>
<tbody>
<tr>
<td>JdbcTemplate, NamedParameter</td>
<td>JDBC Application.</td>
</tr>
<tr>
<td>JdbcTemplate, DataSource</td>
<td></td>
</tr>
<tr>
<td>@EnableJms</td>
<td>JMS Application.</td>
</tr>
<tr>
<td>@EnableCaching</td>
<td>Caching abstraction.</td>
</tr>
<tr>
<td>@Test</td>
<td>JUnit.</td>
</tr>
<tr>
<td>@EnableRabbit</td>
<td>RabbitMQ.</td>
</tr>
<tr>
<td>extends Specification</td>
<td>Spock test.</td>
</tr>
<tr>
<td>@EnableBatchProcessing</td>
<td>Spring Batch.</td>
</tr>
<tr>
<td>@MessageEndpoint @EnableIntegration</td>
<td>Spring Integration.</td>
</tr>
<tr>
<td>@Controller @RestController @EnableWebMvc</td>
<td>Spring MVC + Embedded Tomcat.</td>
</tr>
<tr>
<td>@EnableWebSecurity</td>
<td>Spring Security.</td>
</tr>
<tr>
<td>@EnableTransactionManagement</td>
<td>Spring Transaction Management.</td>
</tr>
</tbody>
</table>

See subclasses of `CompilerAutoConfiguration` in the Spring Boot CLI source code to understand exactly how customizations are applied.
Deduced “grab” Coordinates

Spring Boot extends Groovy’s standard @Grab support by letting you specify a dependency without a group or version (for example, @Grab('freemarker')). Doing so consults Spring Boot’s default dependency metadata to deduce the artifact's group and version.

The default metadata is tied to the version of the CLI that you use. It changes only when you move to a new version of the CLI, putting you in control of when the versions of your dependencies may change. A table showing the dependencies and their versions that are included in the default metadata can be found in the appendix.

Default Import Statements

To help reduce the size of your Groovy code, several import statements are automatically included. Notice how the preceding example refers to @Component, @RestController, and @RequestMapping without needing to use fully-qualified names or import statements.

Many Spring annotations work without using import statements. Try running your application to see what fails before adding imports.

Automatic Main Method

Unlike the equivalent Java application, you do not need to include a public static void main(String[] args) method with your Groovy scripts. A SpringApplication is automatically created, with your compiled code acting as the source.

Custom Dependency Management

By default, the CLI uses the dependency management declared in spring-boot-dependencies when resolving @Grab dependencies. Additional dependency management, which overrides the default dependency management, can be configured by using the @DependencyManagementBom annotation. The annotation’s value should specify the coordinates (groupId:artifactId:version) of one or more Maven BOMs.

For example, consider the following declaration:

```groovy
@DependencyManagementBom("com.example.custom-bom:1.0.0")
```

The preceding declaration picks up custom-bom-1.0.0.pom in a Maven repository under com/example/custom-versions/1.0.0/.

When you specify multiple BOMs, they are applied in the order in which you declare them, as shown in the following example:

```groovy
@DependencyManagementBom(["com.example.custom-bom:1.0.0", "com.example.another-bom:1.0.0"])```
The preceding example indicates that the dependency management in `another-bom` overrides the dependency management in `custom-bom`.

You can use `@DependencyManagementBom` anywhere that you can use `@Grab`. However, to ensure consistent ordering of the dependency management, you can use `@DependencyManagementBom` at most once in your application.

### 7.2.2. Applications with Multiple Source Files

You can use “shell globbing” with all commands that accept file input. Doing so lets you use multiple files from a single directory, as shown in the following example:

```bash
$ spring run *.groovy
```

### 7.2.3. Packaging Your Application

You can use the `jar` command to package your application into a self-contained executable jar file, as shown in the following example:

```bash
$ spring jar my-app.jar *.groovy
```

The resulting jar contains the classes produced by compiling the application and all of the application’s dependencies so that it can then be run by using `java -jar`. The jar file also contains entries from the application’s classpath. You can add and remove explicit paths to the jar by using `--include` and `--exclude`. Both are comma-separated, and both accept prefixes, in the form of “+” and “-”, to signify that they should be removed from the defaults. The default includes are as follows:

```text
public/**, resources/**, static/**, templates/**, META-INF/**, *
```

The default excludes are as follows:

```text
.*, repository/**, build/**, target/**, **/*.jar, **/*.groovy
```

Type `spring help jar` on the command line for more information.

### 7.2.4. 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:

```bash
$ 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:
------------------------
gradele-build - Gradle Config [format:build, build:gradle]
gradele-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 8 and `war` packaging:

```
$ spring init --build=gradle --java-version=1.8 --dependencies=websocket --packaging=war sample-app.zip
Using service at https://start.spring.io
Content saved to 'sample-app.zip'
```

### 7.2.5. 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 (v2.4.2)
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:
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`.

### 7.2.6. Adding Extensions to the CLI

You can add extensions to the CLI by using the `install` command. The command takes one or more sets of artifact coordinates in the format `group:artifact:version`, as shown in the following example:

```
$ spring install com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

In addition to installing the artifacts identified by the coordinates you supply, all of the artifacts’ dependencies are also installed.

To uninstall a dependency, use the `uninstall` command. As with the `install` command, it takes one or more sets of artifact coordinates in the format of `group:artifact:version`, as shown in the following example:

```
$ spring uninstall com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

It uninstalls the artifacts identified by the coordinates you supply and their dependencies.

To uninstall all additional dependencies, you can use the `--all` option, as shown in the following example:

```
$ spring uninstall --all
```

### 7.3. Developing Applications with the Groovy Beans DSL

Spring Framework 4.0 has native support for a `beans{}` “DSL” (borrowed from Grails), and you can embed bean definitions in your Groovy application scripts by using the same format. This is sometimes a good way to include external features like middleware declarations, as shown in the following example:
You can mix class declarations with `beans{}` in the same file as long as they stay at the top level, or, if you prefer, you can put the beans DSL in a separate file.

### 7.4. Configuring the CLI with settings.xml

The Spring Boot CLI uses Aether, Maven’s dependency resolution engine, to resolve dependencies. The CLI makes use of the Maven configuration found in `~/.m2/settings.xml` to configure Aether. The following configuration settings are honored by the CLI:

- **Offline**
- **Mirrors**
- **Servers**
- **Proxies**
- **Profiles**
  - Activation
  - Repositories
- **Active profiles**

See [Maven’s settings documentation](https://maven.apache.org/settings.html) for further information.

### 7.5. What to Read Next

There are some [sample groovy scripts](https://github.com/spring-projects/spring-boot) available from the GitHub repository that you can use to try out the Spring Boot CLI. There is also extensive Javadoc throughout the [source code](https://github.com/spring-projects/spring-boot).
If you find that you reach the limit of the CLI tool, you probably want to look at converting your application to a full Gradle or Maven built “Groovy project”. The next section covers Spring Boot’s "Build tool plugins", which you can use with Gradle or Maven.
Chapter 8. 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 “Using Spring Boot” section first.

8.1. 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 3.2 (or later).

Please refer to the plugin’s documentation to learn more:

- Reference (HTML and PDF)
- API

8.2. 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 6 (6.3 or later). Gradle 5.6.x is also supported but this support is deprecated and will be removed in a future release. Please refer to the plugin’s documentation to learn more:

- Reference (HTML and PDF)
- API

8.3. 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:

```xml
<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:
The “Using Spring Boot” section includes a more complete example of using Apache Ant with spring-boot-antlib.

### 8.3.1. 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>destfile</code></td>
<td>The destination jar file to create</td>
<td>Yes</td>
</tr>
<tr>
<td><code>classes</code></td>
<td>The root directory of Java class files</td>
<td>Yes</td>
</tr>
<tr>
<td><code>start-class</code></td>
<td>The main application class to run</td>
<td>No (the default is the first class found that declares a main method)</td>
</tr>
</tbody>
</table>

The following nested elements can be used with the task:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>resources</code></td>
<td>One or more Resource Collections describing a set of Resources that should be added to the content of the created jar file.</td>
</tr>
<tr>
<td><code>lib</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

#### Examples

This section shows two examples of Ant tasks.
Specify start-class

```xml
<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

```xml
<exejar destfile="target/my-application.jar" classes="target/classes">
  <lib>
    <fileset dir="lib" />
  </lib>
</exejar>
```

8.3.2. 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>classesroot</td>
<td>The root directory of Java class files</td>
<td>Yes <em>(unless mainclass is specified)</em></td>
</tr>
<tr>
<td>mainclass</td>
<td>Can be used to short-circuit the <code>main</code> class search</td>
<td>No</td>
</tr>
<tr>
<td>property</td>
<td>The Ant property that should be set with the result</td>
<td>No <em>(result will be logged if unspecified)</em></td>
</tr>
</tbody>
</table>

Examples

This section contains three examples of using `findmainclass`.

Find and log

```xml
<findmainclass classesroot="target/classes" />
```

Find and set

```xml
<findmainclass classesroot="target/classes" property="main-class" />
```
8.4. 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.

8.4.1. 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.

8.4.2. 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`.

8.4.3. 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.

8.4.4. Example Repackage Implementation

The following example shows a typical repackage implementation:
8.5. 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, you can check out the “how-to” guides.
Chapter 9. “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 out 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.

9.1. Spring Boot Application

This section includes topics relating directly to Spring Boot applications.

9.1.1. 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
```

If you need access to the BeanFactory or the Environment, your FailureAnalyzer can implement BeanFactoryAware or EnvironmentAware respectively.

9.1.2. 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.

### 9.1.3. 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:

```java
```
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
public class EnvironmentPostProcessorExample 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) {
        if (!path.exists()) {
            throw new IllegalArgumentException("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);
        }
    }
}
```

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.

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.
9.1.4. Build an ApplicationContext Hierarchy (Adding a Parent or Root Context)

You can use the ApplicationContext class to create parent/child ApplicationContext hierarchies. See “Fluent Builder API” in the ‘Spring Boot features’ section for more information.

9.1.5. 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 (e.g. 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.

9.2. Properties and Configuration

This section includes topics about setting and reading properties and configuration settings and their interaction with Spring Boot applications.

9.2.1. 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

```java
app.encoding=@project.build.sourceEncoding@
app.java.version=@java.version@
```

Yaml

```yaml
app:
  encoding: "@project.build.sourceEncoding@"
  java:
    version: "@java.version@"
```
Only production configuration is filtered that way (in other words, no filtering is applied on `src/test/resources`).

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`:

```xml
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
  </resource>
</resources>
```

You also need to include the following element inside `<plugins/>`:

```xml
<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>
```

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:

```groovy
processResources {
  expand(project.properties)
}
```
You can then refer to your Gradle project's properties by using placeholders, as shown in the following example:

**Properties**

```properties
app.name=${name}
app.description=${description}
```

**Yaml**

```yaml
app:
  name: "${name}"
  description: "${description}"
```

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: `\${...}`.

### 9.2.2. Externalize the Configuration of SpringApplication

A `SpringApplication` has bean properties (mainly 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**

```properties
spring.main.web-application-type=none
spring.main.banner-mode=off
```

**Yaml**

```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 the values specified with the Java API, with the notable exception of the sources used to create the `ApplicationContext`. Consider the following application:
new SpringApplicationBuilder()
    .bannerMode(Banner.Mode.OFF)
    .sources(demo.MyApp.class)
    .run(args);

Now consider the following configuration:

Properties

```
spring.main.sources=com.acme.Config,com.acme.ExtraConfig
spring.main.banner-mode=console
```

Yaml

```
spring:
  main:
    banner-mode: "console"
```

The actual application now shows the banner (as overridden by configuration) and uses three sources for the ApplicationContext (in the following order): demo.MyApp, com.acme.Config, and com.acme.ExtraConfig.

9.2.3. 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 `.yml` extension are also added to the list.

Spring Boot logs the configuration files that are loaded at the **DEBUG** level and the candidates it has not found at **TRACE** level.

See `ConfigFileApplicationListener` for more detail.
9.2.4. 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

```properties
server.port=${port:8080}
```

Yaml

```yaml
server:
  port: "${port:8080}"
```

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.

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.

9.2.5. 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:

```yaml
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.yml` 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:
See “Working with YAML” in the ‘Spring Boot features’ section for more information about YAML.

### 9.2.6. 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.

### 9.2.7. 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:
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.

The documents are merged in the order in which they are encountered. Later values override earlier values.

9.2.8. Discover Built-in Options for External Properties

Spring Boot binds external properties from application.properties (or .yml 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".
9.3. 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.

9.3.1. 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:

```xml
<properties>
  <servlet-api.version>3.1.0</servlet-api.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>
```

The version of the Servlet API has been overridden as, unlike Tomcat 9 and Undertow 2.0, Jetty 9.4 does not support Servlet 4.0.

The following Gradle example shows how to use Undertow in place of Reactor Netty for Spring WebFlux:
### 9.3.2. 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**

```properties
spring.main.web-application-type=none
```

**Yaml**

```yaml
spring:
  main:
    web-application-type: "none"
```

### 9.3.3. 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.
9.3.4. Use a Random Unassigned HTTP Port

To scan for a free port (using OS natives to prevent clashes) use server.port=0.

9.3.5. 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
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class MyWebIntegrationTests {
    @LocalServerPort
    int port;
    // ...
}
```

@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).

9.3.6. Enable HTTP Response Compression

HTTP response compression is supported by Jetty, Tomcat, 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.

### 9.3.7. Configure SSL

SSL can be configured declaratively by setting the various `server.ssl.*` properties, typically in `application.properties` or `application.yml`. The following example shows setting SSL properties in `application.properties`:

**Properties**

```properties
server.port=8443
server.ssl.key-store=classpath:keystore.jks
server.ssl.key-store-password=secret
server.ssl.key-password=another-secret
```

**Yaml**

```yaml
server:
  port: 8443
  ssl:
    key-store: "classpath:keystore.jks"
    key-store-password: "secret"
    key-password: "another-secret"
```

See [Ssl](#) for details of all of the supported properties.

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.

### 9.3.8. Configure HTTP/2

You can enable HTTP/2 support in your Spring Boot application with the `server.http2.enabled`
configuration property. This support depends on the chosen web server and the application environment, since that protocol is not supported out-of-the-box by all JDK8 releases.

Spring Boot does not advise using h2c, the cleartext version of the HTTP/2 protocol. As a result, the following sections require you to configure SSL first. If you still choose to use h2c, you can check the dedicated section.

HTTP/2 with Tomcat

Spring Boot ships by default with Tomcat 9.0.x which supports HTTP/2 out of the box when using JDK 9 or later. Alternatively, HTTP/2 can be used on JDK 8 if the libtcnative 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.

Starting Tomcat 9.0.x on JDK 8 without that native support logs the following error:

```
```

This error is not fatal, and the application still starts with HTTP/1.1 SSL support.

HTTP/2 with Jetty

For HTTP/2 support, Jetty requires the additional org.eclipse.jetty.http2:http2-server dependency. Now depending on your deployment, you also need to choose other dependencies:

- org.eclipse.jetty:jetty-alpn-java-server for applications running on JDK9+
- org.eclipse.jetty:jetty-alpn-openjdk8-server for applications running on JDK8u252+
- org.eclipse.jetty:jetty-alpn-consycrypt-server and the Conscrypt library with no JDK requirement

HTTP/2 with Reactor Netty

The spring-boot-webflux-starter is using by default Reactor Netty as a server. Reactor Netty can be configured for HTTP/2 using the JDK support with JDK 9 or later. For JDK 8 environments, or for optimal runtime performance, this server also supports HTTP/2 with native libraries. To enable that, your application needs to have an additional dependency.

Spring Boot manages the version for the io.netty:netty-tnative-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**

As of Undertow 1.4.0+, HTTP/2 is supported without any additional requirement on JDK8.

**HTTP/2 Cleartext with supported servers**

To enable HTTP/2 with cleartext support, you need to leave the `server.http2.enabled` property set to `false`, and instead apply a customizer specific to your choice of server:

For Tomcat, we need to add an upgrade protocol:

```java
@Bean
public TomcatConnectorCustomizer connectorCustomizer() {
    return (connector) -> connector.addUpgradeProtocol(new Http2Protocol());
}
```

For Jetty, we need to add a connection factory to the existing connector:

```java
@Bean
public JettyServerCustomizer serverCustomizer() {
    return (server) -> {
        HttpConfiguration configuration = new HttpConfiguration();
        configuration.setSendServerVersion(false);
        Arrays.stream(server.getConnectors())
            .filter(connector -> connector instanceof ServerConnector)
            .map(ServerConnector.class::cast)
            .forEach(connector -> {
                connector.addConnectionFactory(new HTTP2CServerConnectionFactory(configuration));
            });
    };
}
```

For Netty, we need to add `h2c` as a supported protocol:

```java
@Bean
public NettyServerCustomizer serverCustomizer() {
    return (server) -> server.protocol(HttpProtocol.H2C);
}
```

For Undertow, we need to enable the HTTP2 option:
9.3.9. 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.yml`, or environment, etc. 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 doesn’t 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):
### 9.3.10. Add a Servlet, Filter, or Listener to an Application

In a servlet stack application, i.e. 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.

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:

<table>
<thead>
<tr>
<th>Server</th>
<th>Servlet stack</th>
<th>Reactive stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jetty</td>
<td>JettyServletWebServerFactory</td>
<td>JettyReactiveWebServerFactory</td>
</tr>
<tr>
<td>Undertow</td>
<td>UndertowServletWebServerFactory</td>
<td>UndertowReactiveWebServerFactory</td>
</tr>
<tr>
<td>Reactor</td>
<td>N/A</td>
<td>NettyReactiveWebServerFactory</td>
</tr>
</tbody>
</table>
@Bean
class FilterRegistrationBean registration(MyFilter filter) {
    FilterRegistrationBean registration = new FilterRegistrationBean(filter);
    registration.setEnabled(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.

### 9.3.11. 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**

```properties
server.tomcat.basedir=my-tomcat
server.tomcat.accesslog.enabled=true
server.tomcat.accesslog.pattern=%t %a %r %s (%D ms)
```

**Yaml**

```yaml
server:
  tomcat:
    basedir: "my-tomcat"
    accesslog:
      enabled: true
      pattern: "%t %a %r %s (%D ms)"
```

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:
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.

9.3.12. 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**. You can register it as a Servlet Filter in your application by setting `server.forward-headers-strategy` is set to **FRAMEWORK**.

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.

If your application runs in Cloud Foundry or Heroku, 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**

```properties
server.tomcat.remoteip.remote-ip-header=x-your-remote-ip-header
server.tomcat.remoteip.protocol-header=x-your-protocol-header
```

**Yaml**

```yaml
server:
  tomcat:
    remoteip:
      remote-ip-header: "x-your-remote-ip-header"
      protocol-header: "x-your-protocol-header"
```

Tomcat is also configured with a default regular expression that matches internal proxies that are to be trusted. By default, IP addresses in **10/8**, **192.168/16**, **169.254/16** and **127/8** are trusted. You can customize the valve's configuration by adding an entry to **application.properties**, as shown in the following example:

**Properties**

```properties
server.tomcat.remoteip.internal-proxies=192\.168\..\d{1,3}\:\\\d{1,3}
```

**Yaml**

```yaml
server:
  tomcat:
    remoteip:
      internal-proxies: "192\.168\..\d{1,3}\:\\\d{1,3}"
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.

### 9.3.13. 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
@Bean
public WebServerFactoryCustomizer<TomcatServletWebServerFactory> sslConnectorCustomizer() {
    return (tomcat) -> tomcat.addAdditionalTomcatConnectors(createSslConnector());
}

private Connector createSslConnector() {
    Http11NioProtocol protocol = (Http11NioProtocol) connector.getProtocolHandler();
    try {
        URL keystore = ResourceUtils.getURL("keystore");
        URL truststore = ResourceUtils.getURL("truststore");
        connector.setScheme("https");
        connector.setSecure(true);
        connector.setPort(8443);
        protocol.setSSLEnabled(true);
        protocol.setKeystoreFile(keystore.toString());
        protocol.setKeystorePass("changeit");
        protocol.setTruststoreFile(truststore.toString());
        protocol.setTruststorePass("changeit");
        protocol.setKeyAlias("apitester");
        return connector;
    } catch (IOException ex) {
        throw new IllegalStateException("Fail to create ssl connector", ex);
    }
}
```

### 9.3.14. Use Tomcat’s LegacyCookieProcessor

By default, the embedded Tomcat used by Spring Boot does not support "Version 0" of the Cookie format, so you may see the following error:
If at all possible, you should consider updating your code to only store values compliant with later Cookie specifications. If, however, you cannot change the way that cookies are written, you can instead configure Tomcat to use a LegacyCookieProcessor. To switch to the LegacyCookieProcessor, use an WebServerFactoryCustomizer bean that adds a TomcatContextCustomizer, as shown in the following example:

```java
@Bean
public WebServerFactoryCustomizer<TomcatServletWebServerFactory>
cookieProcessorCustomizer() {
    return (factory) -> factory
        .addContextCustomizers((context) -> context.setCookieProcessor(new LegacyCookieProcessor()));
}
```

### 9.3.15. 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 to expose metrics via Micrometer, 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**

```yaml
server:
  tomcat:
    mbeanregistry:
      enabled: true
```

### 9.3.16. Enable Multiple Listeners with Undertow

Add an UndertowBuilderCustomizer to the UndertowServletWebServerFactory and add a listener to the Builder, as shown in the following example:
9.3.17. 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
@Bean
public ServerEndpointExporter serverEndpointExporter() {
    return new 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.

9.4. 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.

9.4.1. 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:
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.

### 9.4.2. 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:

```xml
<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
@XmlRootElement
public class MyThing {
    private String name;
    // .. getters and setters
}
```

JAXB is only available out of the box with Java 8. If you're using a more recent Java generation, add the following dependency to your project:

```xml
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
</dependency>
```
To get the server to render XML instead of JSON, you might have to send an `Accept: text/xml` header (or use a browser).

### 9.4.3. 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

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:

<table>
<thead>
<tr>
<th>Enum</th>
<th>Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>com.fasterxml.jackson.databind.DeserializationFeature</code></td>
<td><code>spring.jackson.deserialization.&lt;feature_name&gt;</code></td>
<td><code>true, false</code></td>
</tr>
<tr>
<td><code>com.fasterxml.jackson.core.JsonGenerator.Feature</code></td>
<td><code>spring.jackson.generator.&lt;feature_name&gt;</code></td>
<td><code>true, false</code></td>
</tr>
<tr>
<td><code>com.fasterxml.jackson.databind.MapperFeature</code></td>
<td><code>spring.jackson.mapper.&lt;feature_name&gt;</code></td>
<td><code>true, false</code></td>
</tr>
<tr>
<td><code>com.fasterxml.jackson.core.JsonParser.Feature</code></td>
<td><code>spring.jackson.parser.&lt;feature_name&gt;</code></td>
<td><code>true, false</code></td>
</tr>
<tr>
<td><code>com.fasterxml.jackson.databind.SerializationFeature</code></td>
<td><code>spring.jackson.serialization.&lt;feature_name&gt;</code></td>
<td><code>true, false</code></td>
</tr>
<tr>
<td><code>com.fasterxml.jackson.annotation.JsonInclude.Include</code></td>
<td><code>spring.jackson.default-property-inclusion</code></td>
<td><code>always, non_null, non_absent, non_default, non_empty</code></td>
</tr>
</tbody>
</table>

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.

### 9.4.4. 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.

### 9.4.5. Handling Multipart File Uploads

Spring Boot embraces the Servlet 3 `javax.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
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.

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.

9.4.6. 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=/acme
```

Yaml

```
spring:
  mvc:
    servlet:
      path: "/acme"
```

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.

9.4.7. 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.

9.4.8. 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 ‘master’ 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

9.5. 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
@Test
@WithMockUser(roles="ADMIN")
public void requestProtectedUrlWithUser() throws Exception {
    mvc
        .perform(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, refer to Spring Security’s reference documentation.

9.6. Jersey

9.6.1. 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 than 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:
9.6.2. 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.

```java
@Component
public class JerseyConfig extends ResourceConfig {
    
    public JerseyConfig() {
        register(Endpoint.class);
        setProperties(Collections.singletonMap("jersey.config.server.response.setStatusOverServerError", true));
    }
}
```

9.7. HTTP Clients

Spring Boot offers a number of starters that work with HTTP clients. This section answers questions related to using them.

9.7.1. 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. The following example configures `HttpComponentsClientRequestFactory` with an `HttpClient` that uses a proxy for all hosts except 192.168.0.5:
static class ProxyCustomizer implements RestTemplateCustomizer {

    @Override
    public void customize(RestTemplate restTemplate) {
        HttpHost proxy = new HttpHost("proxy.example.com");
        HttpClient httpClient = HttpClientBuilder.create().setRoutePlanner(new DefaultProxyRoutePlanner(proxy) {
            @Override
            public HttpHost determineProxy(HttpHost target, HttpRequest request, HttpContext context) throws HttpException {
                if (target.getHostName().equals("192.168.0.5")) {
                    return null;
                }
                return super.determineProxy(target, request, context);
            }
        }).build();
        restTemplate.setRequestFactory(new HttpComponentsClientHttpRequestFactory(httpClient));
    }
}

9.7.2. 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:

@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);
}

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.
9.8. 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:

```xml
<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**

```properties
logging.level.org.springframework.web=debug
logging.level.org.hibernate=error
```

**Yaml**

```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.

9.8.1. Configure Logback for Logging

If you need to apply customizations to logback beyond those that can be achieved with `application.properties`, you'll 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.
The Logback documentation has a dedicated section that covers configuration in some detail.

Spring Boot provides a number of logback configurations that be included from 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
<?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.

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
<?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"
```

### 9.8.2. 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:
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).

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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Dependencies</th>
<th>File names</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAML</td>
<td>com.fasterxml.jackson.core:jackson-databind com.fasterxml.jackson.dataformat:jackson-dataformat-yaml</td>
<td>log4j2.yaml + log4j2.yml</td>
</tr>
<tr>
<td>JSON</td>
<td>com.fasterxml.jackson.core:jackson-databind</td>
<td>log4j2.json + log4j2.jsn</td>
</tr>
</tbody>
</table>

## 9.9. Data Access

Spring Boot includes a number of starters for working with data sources. This section answers questions related to doing so.

### 9.9.1. 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
@Bean
@ConfigurationProperties(prefix="app.datasource")
public DataSource dataSource() {
    return new FancyDataSource();
}
```

The following example shows how to define a data source by setting properties:

**Properties**

```properties
app.datasource.url=jdbc:h2:mem:mydb
app.datasource.username=sa
app.datasource.pool-size=30
```
Yaml

```yaml
app:
  datasource:
    url: "jdbc:h2:mem:mydb"
    username: "sa"
    pool-size: 30
```

Assuming that your FancyDataSource 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. The regular database initialization also happens (so the relevant sub-set of spring.datasource.* can still be used with your custom configuration).

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's 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
@Bean
@ConfigurationProperties("app.datasource")
public 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**

```properties
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.pool-size=30
```

Yaml

```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**

```properties
app.datasource.jdbc-url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.pool-size=30
```

**Yaml**

```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
@Bean
@ConfigurationProperties("app.datasource")
public HikariDataSource dataSource() {
    return DataSourceBuilder.create().type(HikariDataSource.class).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:
@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();
}

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
```

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.

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.
9.9.2. 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
@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSourceProperties firstDataSourceProperties() {
    return new DataSourceProperties();
}

@Bean
@Primary
@ConfigurationProperties("app.datasource.first.configuration")
public HikariDataSource firstDataSource() {
    return firstDataSourceProperties().initializeDataSourceBuilder().type(HikariDataSource.class).build();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public BasicDataSource secondDataSource() {
    return DataSourceBuilder.create().type(BasicDataSource.class).build();
}
```

`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
```

You can apply the same concept to the secondary `DataSource` as well, as shown in the following example:

```java
@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSourceProperties firstDataSourceProperties() {
    return new DataSourceProperties();
}

@Bean
@Primary
@ConfigurationProperties("app.datasource.first.configuration")
public HikariDataSource firstDataSource() {
    return firstDataSourceProperties().initializeDataSourceBuilder().type(HikariDataSource.class).build();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public DataSourceProperties secondDataSourceProperties() {
    return new DataSourceProperties();
}

@Bean
@ConfigurationProperties("app.datasource.second.configuration")
public BasicDataSource secondDataSource() {
    return secondDataSourceProperties().initializeDataSourceBuilder().type(BasicDataSource.class).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.

9.9.3. 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 the same package (or a sub-package) of your @EnableAutoConfiguration class.

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, etc. To get started, create some repository interfaces to handle your @Entity objects.

Spring Boot tries to guess the location of your @Repository definitions, based on the @EnableAutoConfiguration it finds. To get more control, use the @EnableJpaRepositories annotation (from Spring Data JPA).

For more about Spring Data, see the Spring Data project page.

9.9.4. Separate @Entity Definitions from Spring Configuration

Spring Boot tries to guess the location of your @Entity definitions, based on the @EnableAutoConfiguration it finds. To get more control, you can use the @EntityScan annotation, as shown in the following example:

```java
@Configuration(proxyBeanMethods = false)
@EnableAutoConfiguration
@EntityScan(basePackageClasses=City.class)
public class Application {
    //...
}
```

9.9.5. 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**

```properties
spring.jpa.hibernate.naming.physical-strategy=com.example.MyPhysicalNamingStrategy
spring.jpa.show-sql=true
```

**Yaml**

```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.

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.

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.

### 9.9.6. 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 `SpringPhysicalNamingStrategy`. This implementation provides the same table structure as Hibernate 4: 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 `SpringPhysicalNamingStrategy` bean, as shown in the following example:
If you prefer to use Hibernate 5’s default instead, set the following property:

```java
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

Alternatively, you can configure the following bean:

```java
@Bean
public PhysicalNamingStrategy physicalNamingStrategy() {
    return new PhysicalNamingStrategyStandardImpl();
}
```

See [HibernateJpaAutoConfiguration](#) and [JpaBaseConfiguration](#) for more details.

### 9.9.7. 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:hibernate-jcache` is available on the classpath. Then, add a `HibernatePropertiesCustomizer` bean as shown in the following example:

```java
@Configuration(proxyBeanMethods = false)
public class HibernateSecondLevelCacheExample {
    @Bean
    public HibernatePropertiesCustomizer hibernateSecondLevelCacheCustomizer(JCacheCacheManager cacheManager) {
        return (properties) -> properties.put(ConfigSettings.CACHE_MANAGER, cacheManager.getCacheManager());
    }
}
```
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, refer to the Hibernate user guide.

9.9.8. 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 behaviour by registering a `HibernatePropertiesCustomizer` that removes or changes the `hibernate.resource.beans.container` property.

9.9.9. 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.

9.9.10. Use Two EntityManagers

Even if the default `EntityManagerFactory` works fine, you need to define a new one, otherwise the presence of the second bean of that type switches off the default. You can use the `EntityManagerBuilder` provided by Spring Boot to help you to create one. Alternatively, you can use the `LocalContainerEntityManagerFactoryBean` directly from Spring ORM, as shown in the following example:

```java
// add two data sources configured as above

@Bean
public LocalContainerEntityManagerFactoryBean customerEntityManagerFactory(
    EntityManagerFactoryBuilder builder) {
    return builder
        .dataSource(customerDataSource())
        .packages(Customer.class)
        .persistenceUnit("customers")
        .build();
}

@Bean
public LocalContainerEntityManagerFactoryBean orderEntityManagerFactory(
    EntityManagerFactoryBuilder builder) {
    return builder
        .dataSource(orderDataSource())
        .packages(Order.class)
        .persistenceUnit("orders")
        .build();
}
```
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. On the other hand, properties that get applied to the auto-configured `EntityManagerFactoryBuilder`, which are specified via `spring.jpa.properties`, will automatically be applied, provided you use the auto-configured `EntityManagerFactoryBuilder` to build the `LocalContainerEntityManagerFactoryBean` bean.

The configuration above almost works on its own. To complete the picture, you need to configure `TransactionManagers` for the two `EntityManagers` as well. If you mark one of them as @Primary, it could be picked up by the default `JpaTransactionManager` in Spring Boot. The other would have to be explicitly injected into a new instance. 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 example:

```java
@Configuration(proxyBeanMethods = false)
@EnableJpaRepositories(basePackageClasses = Customer.class,
    entityManagerFactoryRef = "customerEntityManagerFactory")
public class CustomerConfiguration {
    ...
}

@Configuration(proxyBeanMethods = false)
@EnableJpaRepositories(basePackageClasses = Order.class,
    entityManagerFactoryRef = "orderEntityManagerFactory")
public class OrderConfiguration {
    ...
}
```

### 9.9.11. 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.

### 9.9.12. 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, Solr, and others). To work with them, change the names of the annotations and
flags accordingly.

9.9.13. 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.

9.9.14. 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.

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.

9.9.15. 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:
9.9.16. Configure jOOQ with Two DataSources

If you need to use jOOQ with multiple data sources, you should create your own DSLContext for each one. Refer to JooqAutoConfiguration for more details.

In particular, JooqExceptionTranslator and SpringTransactionProvider can be reused to provide similar features to what the auto-configuration does with a single DataSource.

9.10. 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.

9.10.1. 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.

9.10.2. 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.

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).

### 9.10.3. Initialize a Database using basic SQL scripts

Spring Boot can automatically create the schema (DDL scripts) of your `DataSource` and initialize it (DML scripts). It loads SQL from the standard root classpath locations: `schema.sql` and `data.sql`, respectively. In addition, Spring Boot processes the `schema-$\{platform\}.sql` and `data-$\{platform\}.sql` files (if present), where `platform` is the value of `spring.datasource.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).

When only basic SQL scripts are used, Spring Boot automatically creates the schema of an embedded `DataSource`. This behavior can be customized by using the `spring.datasource-initialization-mode` property. For instance, if you want to always initialize the `DataSource` regardless of its type:

```properties
spring.datasource.initialization-mode=always
```

In a JPA-based app, you can choose to let Hibernate create the schema or use `schema.sql`, but you cannot do both. Make sure to disable `spring.jpa.hibernate.ddl-auto` if you use `schema.sql`.

```properties
spring.jpa.hibernate.ddl-auto=none
```

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.

By default, Spring Boot enables the fail-fast feature of the Spring JDBC initializer. This means that, if the scripts cause exceptions, the application fails to start. You can tune that behavior by setting `spring.datasource.continue-on-error`.

### 9.10.4. Initialize a Database Using R2DBC

If you are using R2DBC, the regular `DataSource` auto-configuration backs off so none of the options described above can be used.
You can initialize the database on startup using SQL scripts as shown in the following example:

```java
@Configuration(proxyBeanMethods = false)
static class DatabaseInitializationConfiguration {

    @Autowired
    void initializeDatabase(ConnectionFactory connectionFactory) {
        ResourceLoader resourceLoader = new DefaultResourceLoader();
        Resource[] scripts = new Resource[] {
            resourceLoader.getResource("classpath:schema.sql"),
            resourceLoader.getResource("classpath:data.sql")
        };
        new ResourceDatabasePopulator(scripts).populate(connectionFactory).block();
    }
}
```

Alternatively, you can configure either Flyway or Liquibase to configure a DataSource for you for the duration of the migration. Both these libraries offer properties to set the url, username and password of the database to migrate.

When choosing this option, org.springframework:spring-jdbc is still a required dependency.

### 9.10.5. 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**

```properties
spring.batch.initialize-schema=always
```

**Yaml**

```yaml
spring:
  batch:
    initialize-schema: "always"
```

You can also switch off the initialization explicitly by setting spring.batch.initialize-schema to never.

### 9.10.6. 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.

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 (e.g. 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.

9.11. Messaging

Spring Boot offers a number of starters that include messaging. This section answers questions that arise from using messaging with Spring Boot.

9.11.1. 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
@Bean
public DefaultJmsListenerContainerFactory jmsListenerContainerFactory(
    ConnectionFactory connectionFactory,
    DefaultJmsListenerContainerFactoryConfigurer configurer) {
    DefaultJmsListenerContainerFactory listenerFactory =
        new 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.

9.12. Batch Applications

A number of questions often arise when people use Spring Batch from within a Spring Boot application. This section addresses those questions.
9.12.1. 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, implement `BatchConfigurer`. See The Javadoc of `@EnableBatchProcessing` for more details.

For more info about Spring Batch, see the Spring Batch project page.

9.12.2. Running Spring Batch Jobs on Startup

Spring Batch auto-configuration is enabled by adding `@EnableBatchProcessing` to one of your `@Configuration` classes.

By default, it executes all Jobs in the application context on startup (see `JobLauncherApplicationRunner` for details). You can narrow down to a specific job or jobs by specifying `spring.batch.job.names` (which takes a comma-separated list of job name patterns).

See `BatchAutoConfiguration` and `@EnableBatchProcessing` for more details.

9.12.3. 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 (i.e. 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`.

9.12.4. 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.

9.13. Actuator

Spring Boot includes the Spring Boot Actuator. This section answers questions that often arise from its use.
9.13.1. 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.

9.13.2. 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).

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.

9.13.3. Sanitize Sensitive Values

Information returned by the `env` and `configprops` endpoints can be somewhat sensitive so keys matching a certain pattern are sanitized by default (i.e. their values are replaced by `******`).

The patterns to use can be customized using the `management.endpoint.env.keys-to-sanitize` and `management.endpoint.configprops.keys-to-sanitize` respectively.

Spring Boot uses sensible defaults for such keys: any key ending with the word "password", "secret", "key", "token", "vcap_services", "sun.java.command" is entirely sanitized. Additionally, any key that holds the word `credentials` as part of the key is sanitized (configured as a regular expression, i.e. `*credentials.*`).

Furthermore, Spring Boot only sanitizes the sensitive portion of URIs for keys which end with "uri", "uris", "address", or "addresses". The sensitive portion of the URI is identified using the format `<scheme>://<username>:<password>@<host>:<port>/`. For example, for the property `myclient.uri=http://user1:password1@localhost:8081`, the resulting sanitized value is...
9.13.4. 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 via Micrometer. By default, the status codes “UP”, “DOWN”, “OUT_OF_SERVICE” and “UNKNOWN” are used by Spring Boot. To export these, you'll 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
@Configuration
public class HealthMetricsConfiguration {

    public HealthMetricsConfiguration(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;
        } else if (Status.OUT_OF_SERVICE.equals(status)) {
            return 2;
        } else if (Status.DOWN.equals(status)) {
            return 1;
        } else {
            return 0;
        }
    }
}
```


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](#).


If you define a `@Configuration` with a `WebSecurityConfigurerAdapter` or a `SecurityFilterChain` bean in your application, it switches off the default webapp security settings in Spring Boot.
9.14.2. 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.

9.14.3. 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:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    // Customize the application security
    http.requiresChannel().anyRequest().requiresSecure();
    return http.build();
}
```
9.15. Hot Swapping

Spring Boot supports hot swapping. This section answers questions about how it works.

9.15.1. 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.

9.15.2. 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.

9.15.3. 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.

9.15.4. 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.

9.16. Build

Spring Boot includes build plugins for Maven and Gradle. This section answers common questions about these plugins.

9.16.1. 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:

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.4.2</version>
      <executions>
        <execution>
          <goals>
            <goal>build-info</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

See the Spring Boot Maven Plugin documentation for more details.

The following example does the same with Gradle:
### 9.16.2. 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 to your POM:

```xml
<build>
  <plugins>
    <plugin>
      <groupId>pl.project13.maven</groupId>
      <artifactId>git-commit-id-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:

```groovy
plugins {
  id "com.gorylenko.gradle-git-properties" version "2.2.4"
}
```

Both the Maven and Gradle plugins allow the properties that are included in `git.properties` to be configured.

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.

### 9.16.3. 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.
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.

### 9.16.4. 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:

```xml
<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:

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.4.2</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

See the plugin documentation for full usage details.

### 9.16.5. 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:

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

### 9.16.6. 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.

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:

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

9.16.7. 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.yml 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:
9.16.8. 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.

9.16.9. 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:

```xml
<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>
```

9.17. Traditional Deployment

Spring Boot supports traditional deployment as well as more modern forms of deployment. This section answers common questions about traditional deployment.

9.17.1. Create a Deployable War File

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`,

![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.
as shown in the following example:

```java
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)
    {
        return application.sources(Application.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, 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:

```xml
<packaging>war</packaging>
```

If you use Gradle, you need to modify `build.gradle` to apply the war plugin to the project, as follows:

```groovy
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:

```xml
<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:
provided:

```java
dependencies {
    // ...
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    // ...
}
```

`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.

### 9.17.2. 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
@SpringBootApplication
public class Application 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 (via @SpringBootApplication)
        // we actually don't 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(Application.class, args);
}
```
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
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return configureApplication(builder);
    }

    public static void main(String[] args) {
        configureApplication(new SpringApplicationBuilder()).run(args);
    }

    private static SpringApplicationBuilder configureApplication(SpringApplicationBuilder builder) {
        return builder.sources(Application.class).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.

9.17.3. 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 {

}
```

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
<?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>
```

9.17.4. 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:
The following example shows how to do so in Gradle:

```groovy
dependencies {
    implementation('org.springframework.boot:spring-boot-starter-data-redis') {
        exclude group: 'io.lettuce', module: 'lettuce-core'
    }
    implementation 'redis.clients:jedis'
    // ...
}
```

### 9.17.5. Use Testcontainers for integration testing

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 etc. Testcontainers can be used in a Spring Boot test as follows:

```java
@SpringBootTest
@Testcontainers
class ExampleIntegrationTests {
    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>();
}
```

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 using details from the running container, such as container IP or port.

This can be done with a static `@DynamicPropertySource` method that allows adding dynamic property
values to the Spring Environment.

```java
@SpringBootTest
@Testcontainers
class ExampleIntegrationTests {

    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>();

    @DynamicPropertySource
    static void neo4jProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.neo4j.uri", neo4j::getBoltUrl);
    }
}
```

The above configuration allows Neo4j-related beans in the application to communicate with Neo4j running inside the Testcontainers-managed Docker container.
Chapter 10. Appendices

Appendix A: Common Application Properties

Various properties can be specified inside your `application.properties` file, inside your `application.yml` 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.

Spring Boot provides various conversion mechanism with advanced value formatting, make sure to review the properties conversion section.

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.

10.A.1. Core Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>false</td>
<td>Enable debug logs.</td>
</tr>
<tr>
<td>info.*</td>
<td></td>
<td>Arbitrary properties to add to the info endpoint.</td>
</tr>
<tr>
<td>logging.charset.console</td>
<td></td>
<td>Charset to use for console output.</td>
</tr>
<tr>
<td>logging.charset.file</td>
<td></td>
<td>Charset to use for file output.</td>
</tr>
<tr>
<td>logging.config</td>
<td></td>
<td>Location of the logging configuration file. For instance, <code>classpath:logback.xml</code> for Logback.</td>
</tr>
<tr>
<td>logging.exception-conversion-word</td>
<td>%wEx</td>
<td>Conversion word used when logging exceptions.</td>
</tr>
<tr>
<td>logging.file.name</td>
<td></td>
<td>Log file name (for instance, <code>myapp.log</code>). Names can be an exact location or relative to the current directory.</td>
</tr>
<tr>
<td>logging.file.path</td>
<td></td>
<td>Location of the log file. For instance, <code>/var/log</code>.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>---------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 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`.
<p>| logging.logback.rollingpolicy.clean-history-on-start | false | Whether to clean the archive log files on startup. |
| logging.logback.rollingpolicy.file-name-pattern | <code>${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz</code> | Pattern for rolled-over log file names. |
| logging.logback.rollingpolicy.max-file-size | 10MB | Maximum log file size. |
| logging.logback.rollingpolicy.max-history | 7.0 | Maximum number of days archive log files are kept. |
| logging.logback.rollingpolicy.total-size-cap | 0B | Total size of log backups to be kept. |
| logging.pattern.console | %clr(%d{&quot;LOG_DATEFORMAT_AT_PATTERN:=-yyyy-MM-dd HH:mm:ss.SSS}&quot;{faint}) %clr(${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} | Appender pattern for output to the console. Supported only with the default Logback setup. |
| logging.pattern.dateformat | <code>yyyy-MM-dd HH:mm:ss.SSS</code> | Appender pattern for log date format. Supported only with the default Logback setup. |</p>
<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logging.pattern.file</td>
<td>%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}} ${LOG_LEVEL_PATTERN:-%5p} ${PID:- } --- [%t] %-40.40logger{39} : %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}</td>
<td>Appender pattern for output to a file. Supported only with the default Logback setup.</td>
</tr>
<tr>
<td>logging.pattern.level</td>
<td>%5p</td>
<td>Appender pattern for log level. Supported only with the default Logback setup.</td>
</tr>
<tr>
<td>logging.register-shutdown-hook</td>
<td>false</td>
<td>Register a shutdown hook for the logging system when it is initialized.</td>
</tr>
<tr>
<td>spring.aop.auto</td>
<td>true</td>
<td>Add @EnableAspectJAutoProxy.</td>
</tr>
<tr>
<td>spring.aop.proxy-target-class</td>
<td>true</td>
<td>Whether subclass-based (CGLIB) proxies are to be created (true), as opposed to standard Java interface-based proxies (false).</td>
</tr>
<tr>
<td>spring.application.admin.enabled</td>
<td>false</td>
<td>Whether to enable admin features for the application.</td>
</tr>
<tr>
<td>spring.application.admin.jmx-name</td>
<td>org.springframework.boot:type=Admin,name=SpringApplication</td>
<td>JMX name of the application admin MBean.</td>
</tr>
<tr>
<td>spring.application.name</td>
<td></td>
<td>Application name.</td>
</tr>
<tr>
<td>spring.autoconfigure.exclude</td>
<td></td>
<td>Auto-configuration classes to exclude.</td>
</tr>
<tr>
<td>spring.banner.charset</td>
<td>UTF-8</td>
<td>Banner file encoding.</td>
</tr>
<tr>
<td>spring.banner.image.bitdepth</td>
<td>4.0</td>
<td>Bit depth to use for ANSI colors. Supported values are 4 (16 color) or 8 (256 color).</td>
</tr>
<tr>
<td>spring.banner.image.height</td>
<td></td>
<td>Height of the banner image in chars (default based on image height).</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.banner.image.invert</td>
<td>false</td>
<td>Whether images should be inverted for dark terminal themes.</td>
</tr>
<tr>
<td>spring.banner.image.location</td>
<td>classpath:banner.gif</td>
<td>Banner image file location (jpg or png can also be used).</td>
</tr>
<tr>
<td>spring.banner.image.margin</td>
<td>2.0</td>
<td>Left hand image margin in chars.</td>
</tr>
<tr>
<td>spring.banner.image.pixelmode</td>
<td>TEXT</td>
<td>Pixel mode to use when rendering the image.</td>
</tr>
<tr>
<td>spring.banner.image.width</td>
<td>76.0</td>
<td>Width of the banner image in chars.</td>
</tr>
<tr>
<td>spring.banner.location</td>
<td>classpath:banner.txt</td>
<td>Banner text resource location.</td>
</tr>
<tr>
<td>spring.beaninfo.ignore</td>
<td>true</td>
<td>Whether to skip search of BeanInfo classes.</td>
</tr>
<tr>
<td>spring.codec.log-request-details</td>
<td>false</td>
<td>Whether to log form data at DEBUG level, and headers at TRACE level.</td>
</tr>
<tr>
<td>spring.codec.max-in-memory-size</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>spring.config.activate.on-cloud-platform</td>
<td></td>
<td>Required cloud platform for the document to be included.</td>
</tr>
<tr>
<td>spring.config.activate.on-profile</td>
<td></td>
<td>Profile expressions that should match for the document to be included.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.config.additional-location</code></td>
<td></td>
<td>Config file locations used in addition to the defaults.</td>
</tr>
<tr>
<td><code>spring.config.import</code></td>
<td></td>
<td>Import additional config data.</td>
</tr>
<tr>
<td><code>spring.config.location</code></td>
<td></td>
<td>Config file locations that replace the defaults.</td>
</tr>
<tr>
<td><code>spring.config.name</code></td>
<td><code>application</code></td>
<td>Config file name.</td>
</tr>
<tr>
<td><code>spring.config.use-legacy-processing</code></td>
<td><code>false</code></td>
<td>Whether to enable configuration data processing legacy mode.</td>
</tr>
<tr>
<td><code>spring.info.build.encoding</code></td>
<td><code>UTF-8</code></td>
<td>File encoding.</td>
</tr>
<tr>
<td><code>spring.info.build.location</code></td>
<td><code>classpath:META-INF/build-info.properties</code></td>
<td>Location of the generated build-info.properties file.</td>
</tr>
<tr>
<td><code>spring.info.git.encoding</code></td>
<td><code>UTF-8</code></td>
<td>File encoding.</td>
</tr>
<tr>
<td><code>spring.info.git.location</code></td>
<td><code>classpath:git.properties</code></td>
<td>Location of the generated git.properties file.</td>
</tr>
<tr>
<td><code>spring.jmx.default-domain</code></td>
<td></td>
<td>JMX domain name.</td>
</tr>
<tr>
<td><code>spring.jmx.enabled</code></td>
<td><code>false</code></td>
<td>Expose management beans to the JMX domain.</td>
</tr>
<tr>
<td><code>spring.jmx.server</code></td>
<td><code>mbeanServer</code></td>
<td>MBeanServer bean name.</td>
</tr>
<tr>
<td><code>spring.jmx.unique-names</code></td>
<td><code>false</code></td>
<td>Whether unique runtime object names should be ensured.</td>
</tr>
<tr>
<td><code>spring.lifecycle.timeout-per-shutdown-phase</code></td>
<td><code>30s</code></td>
<td>Timeout for the shutdown of any phase (group of SmartLifecycle beans with the same 'phase' value).</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.main.allow-bean-definition-overriding</td>
<td>false</td>
<td>Whether bean definition overriding, by registering a definition with the same name as an existing definition, is allowed.</td>
</tr>
<tr>
<td>spring.main.banner-mode</td>
<td>console</td>
<td>Mode used to display the banner when the application runs.</td>
</tr>
<tr>
<td>spring.main.cloud-platform</td>
<td></td>
<td>Override the Cloud Platform auto-detection.</td>
</tr>
<tr>
<td>spring.main.lazy-initialization</td>
<td>false</td>
<td>Whether initialization should be performed lazily.</td>
</tr>
<tr>
<td>spring.main.log-startup-info</td>
<td>true</td>
<td>Whether to log information about the application when it starts.</td>
</tr>
<tr>
<td>spring.main.register-shutdown-hook</td>
<td>true</td>
<td>Whether the application should have a shutdown hook registered.</td>
</tr>
<tr>
<td>spring.main.sources</td>
<td></td>
<td>Sources (class names, package names, or XML resource locations) to include in the ApplicationContext.</td>
</tr>
<tr>
<td>spring.main.web-application-type</td>
<td></td>
<td>Flag to explicitly request a specific type of web application. If not set, auto-detected based on the classpath.</td>
</tr>
<tr>
<td>spring.mandatory-file-encoding</td>
<td></td>
<td>Expected character encoding the application must use.</td>
</tr>
<tr>
<td>spring.messages.always-use-message-format</td>
<td>false</td>
<td>Whether to always apply the MessageFormat rules, parsing even messages without arguments.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.messages.basename</td>
<td>messages</td>
<td>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 &quot;org.mypackage&quot;), it will be resolved from the classpath root.</td>
</tr>
<tr>
<td>spring.messages.cache-duration</td>
<td></td>
<td>Loaded resource bundle files cache duration. When not set, bundles are cached forever. If a duration suffix is not specified, seconds will be used.</td>
</tr>
<tr>
<td>spring.messages.encoding</td>
<td>UTF-8</td>
<td>Message bundles encoding.</td>
</tr>
<tr>
<td>spring.messages.fallback-to-system-locale</td>
<td>true</td>
<td>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. &quot;messages.properties&quot; for basename &quot;messages&quot;).</td>
</tr>
<tr>
<td>spring.messages.use-code-as-default-message</td>
<td>false</td>
<td>Whether to use the message code as the default message instead of throwing a &quot;NoSuchMessageException&quot;. Recommended during development only.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.output.ansi.enabled</td>
<td>detect</td>
<td>Configures the ANSI output.</td>
</tr>
<tr>
<td>spring.pid.fail-on-write-error</td>
<td></td>
<td>Fails if ApplicationPidFileWriter is used but it cannot write the PID file.</td>
</tr>
<tr>
<td>spring.pid.file</td>
<td></td>
<td>Location of the PID file to write (if ApplicationPidFileWriter is used).</td>
</tr>
<tr>
<td>spring.profiles.active</td>
<td></td>
<td>Comma-separated list of active profiles. Can be overridden by a command line switch.</td>
</tr>
<tr>
<td>spring.profiles.include</td>
<td></td>
<td>Unconditionally activate the specified comma-separated list of profiles (or list of profiles if using YAML).</td>
</tr>
<tr>
<td>spring.quartz.auto-startup</td>
<td>true</td>
<td>Whether to automatically start the scheduler after initialization.</td>
</tr>
<tr>
<td>spring.quartz.jdbc.comment-prefix</td>
<td>[#,-]</td>
<td>Prefixes for single-line comments in SQL initialization scripts.</td>
</tr>
<tr>
<td>spring.quartz.jdbc.initialize-schema</td>
<td>embedded</td>
<td>Database schema initialization mode.</td>
</tr>
<tr>
<td>spring.quartz.jdbc.schema</td>
<td>classpath:org/quartz/impl/jdbcjobstore/tables_@@platform@@.sql</td>
<td>Path to the SQL file to use to initialize the database schema.</td>
</tr>
<tr>
<td>spring.quartz.job-store-type</td>
<td>memory</td>
<td>Quartz job store type.</td>
</tr>
<tr>
<td>spring.quartz.overwrite-existing-jobs</td>
<td>false</td>
<td>Whether configured jobs should overwrite existing job definitions.</td>
</tr>
<tr>
<td>spring.quartz.properties.*</td>
<td></td>
<td>Additional Quartz Scheduler properties.</td>
</tr>
<tr>
<td>spring.quartz.scheduler-name</td>
<td>quartzScheduler</td>
<td>Name of the scheduler.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.quartz.startup-delay</code></td>
<td>0s</td>
<td>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.</td>
</tr>
<tr>
<td><code>spring.quartz.wait-for-jobs-to-complete-on-shutdown</code></td>
<td>false</td>
<td>Whether to wait for running jobs to complete on shutdown.</td>
</tr>
<tr>
<td><code>spring.reactor.debug-agent.enabled</code></td>
<td>true</td>
<td>Whether the Reactor Debug Agent should be enabled when reactor-tools is present.</td>
</tr>
<tr>
<td><code>spring.task.execution.pool.allow-core-thread-timeout</code></td>
<td>true</td>
<td>Whether core threads are allowed to time out. This enables dynamic growing and shrinking of the pool.</td>
</tr>
<tr>
<td><code>spring.task.execution.pool.core-size</code></td>
<td>8.0</td>
<td>Core number of threads.</td>
</tr>
<tr>
<td><code>spring.task.execution.pool.keep-alive</code></td>
<td>60s</td>
<td>Time limit for which threads may remain idle before being terminated.</td>
</tr>
<tr>
<td><code>spring.task.execution.pool.max-size</code></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><code>spring.task.execution.pool.queue-capacity</code></td>
<td></td>
<td>Queue capacity. An unbounded capacity does not increase the pool and therefore ignores the &quot;max-size&quot; property.</td>
</tr>
</tbody>
</table>
### 10.A.2. Cache Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.cache.cache-names</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>spring.cache.caffeine.spec</td>
<td></td>
<td>The spec to use to create caches. See CaffeineSpec for more details on the spec format.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.cache.couchbase.expiration</code></td>
<td></td>
<td>Entry expiration. By default the entries never expire. Note that this value is ultimately converted to seconds.</td>
</tr>
<tr>
<td><code>spring.cache.ehcache.config</code></td>
<td></td>
<td>The location of the configuration file to use to initialize EhCache.</td>
</tr>
<tr>
<td><code>spring.cache.infinispan.config</code></td>
<td></td>
<td>The location of the configuration file to use to initialize Infinispan.</td>
</tr>
<tr>
<td><code>spring.cache.jcache.config</code></td>
<td></td>
<td>The location of the configuration file to use to initialize the cache manager. The configuration file is dependent of the underlying cache implementation.</td>
</tr>
<tr>
<td><code>spring.cache.jcache.provider</code></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><code>spring.cache.redis.cache-null-values</code></td>
<td><code>true</code></td>
<td>Allow caching null values.</td>
</tr>
<tr>
<td><code>spring.cache.redis.enable-statistics</code></td>
<td><code>false</code></td>
<td>Whether to enable cache statistics.</td>
</tr>
<tr>
<td><code>spring.cache.redis.key-prefix</code></td>
<td></td>
<td>Key prefix.</td>
</tr>
<tr>
<td><code>spring.cache.redis.time-to-live</code></td>
<td></td>
<td>Entry expiration. By default the entries never expire.</td>
</tr>
<tr>
<td><code>spring.cache.redis.use-key-prefix</code></td>
<td><code>true</code></td>
<td>Whether to use the key prefix when writing to Redis.</td>
</tr>
</tbody>
</table>
### 10.A.3. Mail Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.mail.default-encoding</td>
<td>UTF-8</td>
<td>Default MimeMessage encoding.</td>
</tr>
<tr>
<td>spring.mail.host</td>
<td></td>
<td>SMTP server host. For instance, <code>smtp.example.com</code>.</td>
</tr>
<tr>
<td>spring.mail.jndi-name</td>
<td></td>
<td>Session JNDI name. When set, takes precedence over other Session settings.</td>
</tr>
<tr>
<td>spring.mail.password</td>
<td></td>
<td>Login password of the SMTP server.</td>
</tr>
<tr>
<td>spring.mail.port</td>
<td></td>
<td>SMTP server port.</td>
</tr>
<tr>
<td>spring.mail.properties.*</td>
<td></td>
<td>Additional JavaMail Session properties.</td>
</tr>
<tr>
<td>spring.mail.protocol</td>
<td>smtp</td>
<td>Protocol used by the SMTP server.</td>
</tr>
<tr>
<td>spring.mail.test-connection</td>
<td>false</td>
<td>Whether to test that the mail server is available on startup.</td>
</tr>
<tr>
<td>spring.mail.username</td>
<td></td>
<td>Login user of the SMTP server.</td>
</tr>
<tr>
<td>spring.sendgrid.api-key</td>
<td></td>
<td>SendGrid API key.</td>
</tr>
<tr>
<td>spring.sendgrid.proxy.host</td>
<td></td>
<td>SendGrid proxy host.</td>
</tr>
<tr>
<td>spring.sendgrid.proxy.port</td>
<td></td>
<td>SendGrid proxy port.</td>
</tr>
</tbody>
</table>

### 10.A.4. JSON Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.gson.date-format</td>
<td></td>
<td>Format to use when serializing Date objects.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.gson.disable-html-escaping</td>
<td></td>
<td>Whether to disable the escaping of HTML characters such as '&amp;', '&gt;', etc.</td>
</tr>
<tr>
<td>spring.gson.disable-inner-class-serialization</td>
<td></td>
<td>Whether to exclude inner classes during serialization.</td>
</tr>
<tr>
<td>spring.gson.enable-complex-map-key-serialization</td>
<td></td>
<td>Whether to enable serialization of complex map keys (i.e. non-primitives).</td>
</tr>
<tr>
<td>spring.gson.exclude-fields-without-expose-annotation</td>
<td></td>
<td>Whether to exclude all fields from consideration for serialization or deserialization that do not have the &quot;Expose&quot; annotation.</td>
</tr>
<tr>
<td>spring.gson.field-naming-policy</td>
<td></td>
<td>Naming policy that should be applied to an object's field during serialization and deserialization.</td>
</tr>
<tr>
<td>spring.gson.generate-non-executable-json</td>
<td></td>
<td>Whether to generate non executable JSON by prefixing the output with some special text.</td>
</tr>
<tr>
<td>spring.gson.lenient</td>
<td></td>
<td>Whether to be lenient about parsing JSON that doesn't conform to RFC 4627.</td>
</tr>
<tr>
<td>spring.gson.long-serialization-policy</td>
<td></td>
<td>Serialization policy for Long and long types.</td>
</tr>
<tr>
<td>spring.gson.pretty-printing</td>
<td></td>
<td>Whether to output serialized JSON that fits in a page for pretty printing.</td>
</tr>
<tr>
<td>spring.gson.serialize-nulls</td>
<td></td>
<td>Whether to serialize null fields.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>----------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jackson.date-format</td>
<td></td>
<td>Date format string or a fully-qualified date format class name. For instance, <code>yyyy-MM-dd HH:mm:ss</code>.</td>
</tr>
<tr>
<td>spring.jackson.default-property-inclusion</td>
<td></td>
<td>Controls the inclusion of properties during serialization. Configured with one of the values in Jackson's JsonInclude.Include enumeration.</td>
</tr>
<tr>
<td>spring.jackson.deserialization.*</td>
<td></td>
<td>Jackson on/off features that affect the way Java objects are deserialized.</td>
</tr>
<tr>
<td>spring.jackson.generator.*</td>
<td></td>
<td>Jackson on/off features for generators.</td>
</tr>
<tr>
<td>spring.jackson.locale</td>
<td></td>
<td>Locale used for formatting.</td>
</tr>
<tr>
<td>spring.jackson.mapper.*</td>
<td></td>
<td>Jackson general purpose on/off features.</td>
</tr>
<tr>
<td>spring.jackson.parser.*</td>
<td></td>
<td>Jackson on/off features for parsers.</td>
</tr>
<tr>
<td>spring.jackson.property-naming-strategy</td>
<td></td>
<td>One of the constants on Jackson's PropertyNamingStrategy. Can also be a fully-qualified class name of a PropertyNamingStrategy subclass.</td>
</tr>
<tr>
<td>spring.jackson.serialization.*</td>
<td></td>
<td>Jackson on/off features that affect the way Java objects are serialized.</td>
</tr>
<tr>
<td>spring.jackson.time-zone</td>
<td></td>
<td>Time zone used when formatting dates. For instance, &quot;America/Los_Angeles&quot; or &quot;GMT+10&quot;.</td>
</tr>
</tbody>
</table>
### Spring Jackson Visibility

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.jackson.visibility.*</td>
<td></td>
<td>Jackson visibility thresholds that can be used to limit which methods (and fields) are auto-detected.</td>
</tr>
</tbody>
</table>

### 10.A.5. Data Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.couchbase.connection-string</td>
<td></td>
<td>Connection string used to locate the Couchbase cluster.</td>
</tr>
<tr>
<td>spring.couchbase.env.io.idle-http-connection-timeout</td>
<td>4500ms</td>
<td>Length of time an HTTP connection may remain idle before it is closed and removed from the pool.</td>
</tr>
<tr>
<td>spring.couchbase.env.io.max-endpoints</td>
<td>12.0</td>
<td>Maximum number of sockets per node.</td>
</tr>
<tr>
<td>spring.couchbase.env.io.min-endpoints</td>
<td>1.0</td>
<td>Minimum number of sockets per node.</td>
</tr>
<tr>
<td>spring.couchbase.env.ssl.enabled</td>
<td></td>
<td>Whether to enable SSL support. Enabled automatically if a &quot;keyStore&quot; is provided unless specified otherwise.</td>
</tr>
<tr>
<td>spring.couchbase.env.ssl.key-store</td>
<td></td>
<td>Path to the JVM key store that holds the certificates.</td>
</tr>
<tr>
<td>spring.couchbase.env.ssl.key-store-password</td>
<td></td>
<td>Password used to access the key store.</td>
</tr>
<tr>
<td>spring.couchbase.env.timeouts.analytics</td>
<td>75s</td>
<td>Timeout for the analytics service.</td>
</tr>
<tr>
<td>spring.couchbase.env.timeouts.connect</td>
<td>10s</td>
<td>Bucket connect timeout.</td>
</tr>
<tr>
<td>spring.couchbase.env.timeouts.disconnect</td>
<td>10s</td>
<td>Bucket disconnect timeout.</td>
</tr>
<tr>
<td>spring.couchbase.env.timeouts.key-value</td>
<td>2500ms</td>
<td>Timeout for operations on a specific key-value.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.couchbase.env.timeouts.key-value-durable</code></td>
<td>10s</td>
<td>Timeout for operations on a specific key-value with a durability level.</td>
</tr>
<tr>
<td><code>spring.couchbase.env.timeouts.management</code></td>
<td>75s</td>
<td>Timeout for the management operations.</td>
</tr>
<tr>
<td><code>spring.couchbase.env.timeouts.query</code></td>
<td>75s</td>
<td>N1QL query operations timeout.</td>
</tr>
<tr>
<td><code>spring.couchbase.env.timeouts.search</code></td>
<td>75s</td>
<td>Timeout for the search service.</td>
</tr>
<tr>
<td><code>spring.couchbase.env.timeouts.view</code></td>
<td>75s</td>
<td>Regular and geospatial view operations timeout.</td>
</tr>
<tr>
<td><code>spring.couchbase.password</code></td>
<td></td>
<td>Cluster password.</td>
</tr>
<tr>
<td><code>spring.couchbase.username</code></td>
<td></td>
<td>Cluster username.</td>
</tr>
<tr>
<td><code>spring.dao.exceptiontranslation.enabled</code></td>
<td>true</td>
<td>Whether to enable the PersistenceExceptionTranslationPostProcessor.</td>
</tr>
<tr>
<td><code>spring.data.cassandra.compression</code></td>
<td>none</td>
<td>Compression supported by the Cassandra binary protocol.</td>
</tr>
<tr>
<td><code>spring.data.cassandra.connection.connect-timeout</code></td>
<td>5s</td>
<td>Timeout to use when establishing driver connections.</td>
</tr>
<tr>
<td><code>spring.data.cassandra.connection.init-query-timeout</code></td>
<td>500ms</td>
<td>Timeout to use for internal queries that run as part of the initialization process, just after a connection is opened.</td>
</tr>
<tr>
<td><code>spring.data.cassandra.contact-points</code></td>
<td>[127.0.0.1:9042]</td>
<td>Cluster node addresses in the form 'host:port', or a simple 'host' to use the configured port.</td>
</tr>
<tr>
<td><code>spring.data.cassandra.keyspace-name</code></td>
<td></td>
<td>Keyspace name to use.</td>
</tr>
<tr>
<td><code>spring.data.cassandra.local-datacenter</code></td>
<td></td>
<td>Datacenter that is considered &quot;local&quot;. Contact points should be from this datacenter.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.data.cassandra.password</td>
<td></td>
<td>Login password of the server.</td>
</tr>
<tr>
<td>spring.data.cassandra.pool.heartbeat-interval</td>
<td>30s</td>
<td>Heartbeat interval after which a message is sent on an idle connection to make sure it's still alive.</td>
</tr>
<tr>
<td>spring.data.cassandra.pool.idle-timeout</td>
<td>120s</td>
<td>Idle timeout before an idle connection is removed.</td>
</tr>
<tr>
<td>spring.data.cassandra.port</td>
<td>9042.0</td>
<td>Port to use if a contact point does not specify one.</td>
</tr>
<tr>
<td>spring.data.cassandra.repositories.type</td>
<td>auto</td>
<td>Type of Cassandra repositories to enable.</td>
</tr>
<tr>
<td>spring.data.cassandra.request.consistency</td>
<td></td>
<td>Queries consistency level.</td>
</tr>
<tr>
<td>spring.data.cassandra.request.page-size</td>
<td>5000.0</td>
<td>How many rows will be retrieved simultaneously in a single network roundtrip.</td>
</tr>
<tr>
<td>spring.data.cassandra.request.serial-consistency</td>
<td></td>
<td>Queries serial consistency level.</td>
</tr>
<tr>
<td>spring.data.cassandra.request.throttler.drain-interval</td>
<td>10ms</td>
<td>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.</td>
</tr>
<tr>
<td>spring.data.cassandra.request.throttler.max-concurrent-requests</td>
<td>10000.0</td>
<td>Maximum number of requests that are allowed to execute in parallel.</td>
</tr>
<tr>
<td>spring.data.cassandra.request.throttler.max-queue-size</td>
<td>10000.0</td>
<td>Maximum number of requests that can be enqueued when the throttling threshold is exceeded.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>spring.data.cassandra.request.throttler.max-requests-per-second</td>
<td>10000.0</td>
<td>Maximum allowed request rate.</td>
</tr>
<tr>
<td>spring.data.cassandra.request.throttler.type</td>
<td>none</td>
<td>Request throttling type.</td>
</tr>
<tr>
<td>spring.data.cassandra.request.timeout</td>
<td>2s</td>
<td>How long the driver waits for a request to complete.</td>
</tr>
<tr>
<td>spring.data.cassandra.schema-action</td>
<td>none</td>
<td>Schema action to take at startup.</td>
</tr>
<tr>
<td>spring.data.cassandra.session-name</td>
<td></td>
<td>Name of the Cassandra session.</td>
</tr>
<tr>
<td>spring.data.cassandra.ssl</td>
<td>false</td>
<td>Enable SSL support.</td>
</tr>
<tr>
<td>spring.data.cassandra.username</td>
<td></td>
<td>Login user of the server.</td>
</tr>
<tr>
<td>spring.data.couchbase.auto-index</td>
<td>false</td>
<td>Automatically create views and indexes. Use the meta-data provided by &quot;@ViewIndexed&quot;, &quot;@N1qlPrimaryIndexed&quot; and &quot;@N1qlSecondaryIndexed&quot;.</td>
</tr>
<tr>
<td>spring.data.couchbase.bucket-name</td>
<td></td>
<td>Name of the bucket to connect to.</td>
</tr>
<tr>
<td>spring.data.couchbase.field-naming-strategy</td>
<td></td>
<td>Fully qualified name of the FieldNamingStrategy to use.</td>
</tr>
<tr>
<td>spring.data.couchbase.repositories.type</td>
<td>auto</td>
<td>Type of Couchbase repositories to enable.</td>
</tr>
<tr>
<td>spring.data.couchbase.scope-name</td>
<td></td>
<td>Name of the scope used for all collection access.</td>
</tr>
<tr>
<td>spring.data.couchbase.type-key</td>
<td>_class</td>
<td>Name of the field that stores the type information for complex types when using &quot;MappingCouchbaseConverter&quot;.</td>
</tr>
<tr>
<td>spring.data.elasticsearch.client.reactive.connection-timeout</td>
<td></td>
<td>Connection timeout.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.data.elasticsearch.client.reactive.endpoints</td>
<td></td>
<td>Comma-separated list of the Elasticsearch endpoints to connect to.</td>
</tr>
<tr>
<td>spring.data.elasticsearch.client.reactive.max-in-memory-size</td>
<td></td>
<td>Limit on the number of bytes that can be buffered whenever the input stream needs to be aggregated.</td>
</tr>
<tr>
<td>spring.data.elasticsearch.client.reactive.password</td>
<td></td>
<td>Credentials password.</td>
</tr>
<tr>
<td>spring.data.elasticsearch.client.reactive.socket-timeout</td>
<td></td>
<td>Read and Write Socket timeout.</td>
</tr>
<tr>
<td>spring.data.elasticsearch.client.reactive.use-ssl</td>
<td>false</td>
<td>Whether the client should use SSL to connect to the endpoints.</td>
</tr>
<tr>
<td>spring.data.elasticsearch.client.reactive.username</td>
<td></td>
<td>Credentials username.</td>
</tr>
<tr>
<td>spring.data.elasticsearch.repositories.enabled</td>
<td>true</td>
<td>Whether to enable Elasticsearch repositories.</td>
</tr>
<tr>
<td>spring.data.jdbc.repositories.enabled</td>
<td>true</td>
<td>Whether to enable JDBC repositories.</td>
</tr>
<tr>
<td>spring.data.jpa.repositories.bootstrap-mode</td>
<td>default</td>
<td>Bootstrap mode for JPA repositories.</td>
</tr>
<tr>
<td>spring.data.jpa.repositories.enabled</td>
<td>true</td>
<td>Whether to enable JPA repositories.</td>
</tr>
<tr>
<td>spring.data.ldap.repositories.enabled</td>
<td>true</td>
<td>Whether to enable LDAP repositories.</td>
</tr>
<tr>
<td>spring.data.mongodb.authentication-database</td>
<td></td>
<td>Authentication database name.</td>
</tr>
<tr>
<td>spring.data.mongodb.auto-index-creation</td>
<td></td>
<td>Whether to enable auto-index creation.</td>
</tr>
<tr>
<td>spring.data.mongodb.database</td>
<td></td>
<td>Database name.</td>
</tr>
<tr>
<td>spring.data.mongodb.field-naming-strategy</td>
<td></td>
<td>Fully qualified name of the FieldNamingStrategy to use.</td>
</tr>
<tr>
<td>spring.data.mongodb.gridfs.bucket</td>
<td></td>
<td>GridFS bucket name.</td>
</tr>
<tr>
<td>spring.data.mongodb.gridfs.database</td>
<td></td>
<td>GridFS database name.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.data.mongodb.host</td>
<td></td>
<td>Mongo server host. Cannot be set with URI.</td>
</tr>
<tr>
<td>spring.data.mongodb.password</td>
<td></td>
<td>Login password of the mongo server. Cannot be set with URI.</td>
</tr>
<tr>
<td>spring.data.mongodb.port</td>
<td></td>
<td>Mongo server port. Cannot be set with URI.</td>
</tr>
<tr>
<td>spring.data.mongodb.replica-set-name</td>
<td></td>
<td>Required replica set name for the cluster. Cannot be set with URI.</td>
</tr>
<tr>
<td>spring.data.mongodb.repositories.type</td>
<td>auto</td>
<td>Type of Mongo repositories to enable.</td>
</tr>
<tr>
<td>spring.data.mongodb.uri</td>
<td>mongodb://localhost/test</td>
<td>Mongo database URI. Cannot be set with host, port, credentials and replica set name.</td>
</tr>
<tr>
<td>spring.data.mongodb.username</td>
<td></td>
<td>Login user of the mongo server. Cannot be set with URI.</td>
</tr>
<tr>
<td>spring.data.mongodb.uuid-representation</td>
<td>java-legacy</td>
<td>Representation to use when converting a UUID to a BSON binary value.</td>
</tr>
<tr>
<td>spring.data.neo4j.database</td>
<td></td>
<td>Database name to use. By default, the server decides the default database to use.</td>
</tr>
<tr>
<td>spring.data.neo4j.repositories.type</td>
<td>auto</td>
<td>Type of Neo4j repositories to enable.</td>
</tr>
<tr>
<td>spring.data.r2dbc.repositories.enabled</td>
<td>true</td>
<td>Whether to enable R2DBC repositories.</td>
</tr>
<tr>
<td>spring.data.redis.repositories.enabled</td>
<td>true</td>
<td>Whether to enable Redis repositories.</td>
</tr>
<tr>
<td>spring.data.rest.base-path</td>
<td></td>
<td>Base path to be used by Spring Data REST to expose repository resources.</td>
</tr>
<tr>
<td>spring.data.rest.default-media-type</td>
<td></td>
<td>Content type to use as a default when none is specified.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.data.rest.default-page-size</td>
<td></td>
<td>Default size of pages.</td>
</tr>
<tr>
<td>spring.data.rest.detection-strategy</td>
<td>default</td>
<td>Strategy to use to determine which repositories get exposed.</td>
</tr>
<tr>
<td>spring.data.rest.enable-enum-translation</td>
<td></td>
<td>Whether to enable enum value translation through the Spring Data REST default resource bundle.</td>
</tr>
<tr>
<td>spring.data.rest.limit-param-name</td>
<td></td>
<td>Name of the URL query string parameter that indicates how many results to return at once.</td>
</tr>
<tr>
<td>spring.data.rest.max-page-size</td>
<td></td>
<td>Maximum size of pages.</td>
</tr>
<tr>
<td>spring.data.rest.page-param-name</td>
<td></td>
<td>Name of the URL query string parameter that indicates what page to return.</td>
</tr>
<tr>
<td>spring.data.rest.return-body-on-create</td>
<td></td>
<td>Whether to return a response body after creating an entity.</td>
</tr>
<tr>
<td>spring.data.rest.return-body-on-update</td>
<td></td>
<td>Whether to return a response body after updating an entity.</td>
</tr>
<tr>
<td>spring.data.rest.sort-param-name</td>
<td></td>
<td>Name of the URL query string parameter that indicates what direction to sort results.</td>
</tr>
<tr>
<td>spring.data.solr.host</td>
<td><a href="http://127.0.0.1:8983/solr">http://127.0.0.1:8983/solr</a></td>
<td>Solr host. Ignored if &quot;zk-host&quot; is set.</td>
</tr>
<tr>
<td>spring.data.solr.repositories.enabled</td>
<td>true</td>
<td>Whether to enable Solr repositories.</td>
</tr>
<tr>
<td>spring.data.solr.zk-host</td>
<td></td>
<td>ZooKeeper host address in the form HOST:PORT.</td>
</tr>
<tr>
<td>spring.data.web.pageable.default-page-size</td>
<td>20.0</td>
<td>Default page size.</td>
</tr>
<tr>
<td>spring.data.web.pageable.max-page-size</td>
<td>2000.0</td>
<td>Maximum page size to be accepted.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.data.web.pageable.one-indexed-parameters</td>
<td>false</td>
<td>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.</td>
</tr>
<tr>
<td>spring.data.web.pageable.page-parameter</td>
<td>page</td>
<td>Page index parameter name.</td>
</tr>
<tr>
<td>spring.data.web.pageable.prefix</td>
<td></td>
<td>General prefix to be prepended to the page number and page size parameters.</td>
</tr>
<tr>
<td>spring.data.web.pageable.qualifier-delimiter</td>
<td>_</td>
<td>Delimiter to be used between the qualifier and the actual page number and size properties.</td>
</tr>
<tr>
<td>spring.data.web.pageable.size-parameter</td>
<td>size</td>
<td>Page size parameter name.</td>
</tr>
<tr>
<td>spring.data.web.sort.sort-parameter</td>
<td>sort</td>
<td>Sort parameter name.</td>
</tr>
<tr>
<td>spring.datasource.continue-on-error</td>
<td>false</td>
<td>Whether to stop if an error occurs while initializing the database.</td>
</tr>
<tr>
<td>spring.datasource.data</td>
<td></td>
<td>Data (DML) script resource references.</td>
</tr>
<tr>
<td>spring.datasource.data-password</td>
<td></td>
<td>Password of the database to execute DML scripts (if different).</td>
</tr>
<tr>
<td>spring.datasource.data-username</td>
<td></td>
<td>Username of the database to execute DML scripts (if different).</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.datasource.dbcp2.abandoned-usage-tracking</td>
<td></td>
<td>Commons DBCP2 specific settings bound to an instance of DBCP2's BasicDataSource</td>
</tr>
<tr>
<td>spring.datasource.dbcp2.access-to-underlying-connection-allowed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.auto-commit-on-return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.cache-state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.clear-statement-pool-on-return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.connection-factory-class-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.connection-init-sqls</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.default-auto-commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.default-catalog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.default-query-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.default-read-only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.default-schema</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.default-transaction-isolation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.disconnection-sql-codes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.driver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.driver-class-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.evacuation-policy-class-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.fast-fail-validation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.initial-size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.jmx-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.lifo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.log-abandoned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.log-expired-connections</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.login-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.max-conn-lifetime-millis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.max-idle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.max-open-prepared-statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.max-total</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.max-wait-millis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.min-evictable-idle-time-millis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.min-idle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.num-tests-per-eviction-run</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.password</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.pool-prepared-statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.dbcp2.remove-abandoned-on-reset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.datasource.driver-class-name</td>
<td></td>
<td>Fully qualified name of the JDBC driver. Auto-detected based on the URL by default.</td>
</tr>
<tr>
<td>spring.datasource.generate-unique-name</td>
<td>true</td>
<td>Whether to generate a random datasource name.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.datasource.hikari.allow-pool-suspension</td>
<td></td>
<td>Hikari specific settings bound to an instance of Hikari's HikariDataSource</td>
</tr>
<tr>
<td>spring.datasource.hikari.auto-commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.catalog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.connection-init-sql</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.connection-test-query</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.connection-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.data-source-class-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.data-source-j-n-d-i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.data-source-properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.driver-class-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.exception-override-class-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.health-check-properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.health-check-registry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.idle-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.initialization-fail-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.isolate-internal-queries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.jdbc-url</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.leak-detection-threshold</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.login-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.max-lifetime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.maximum-pool-size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.metric-registry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.metrics-tracker-factory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.minimum-idle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.password</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.pool-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.read-only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.register-j-mbeans</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.scheduled-executor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.schema</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.transaction-isolation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.username</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.hikari.validation-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.datasource.initialization-mode</td>
<td>embedded</td>
<td>Mode to apply when determining if DataSource initialization should be performed using the available DDL and DML scripts.</td>
</tr>
<tr>
<td>spring.datasource.jndi-name</td>
<td></td>
<td>JNDI location of the datasource. Class, url, username and password are ignored when set.</td>
</tr>
<tr>
<td>spring.datasource.name</td>
<td></td>
<td>Name of the datasource. Default to &quot;testdb&quot; when using an embedded database.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.datasource.oracleucp.abandoned-connection-timeout</td>
<td></td>
<td>Oracle UCP specific settings bound to an instance of Oracle UCP’s PoolDataSource</td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-factory-class-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-factory-properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-harvest-max-count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-harvest-trigger-count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-labeling-high-cost</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-pool-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-repurpose-threshold</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-validation-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.connection-wait-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.data-source-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.database-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.fast-connection-failover-enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.high-cost-connection-reuse-threshold</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.inactive-connection-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.initial-pool-size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.login-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.max-connection-reuse-count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.max-connection-reuse-time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.max-connections-per-shard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.max-idle-time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.max-pool-size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.max-statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.min-pool-size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.network-protocol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.o-n-s-configuration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.password</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.oracleucp.pdb-roles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.datasource.password</td>
<td></td>
<td>Login password of the database.</td>
</tr>
<tr>
<td>spring.datasource.platform</td>
<td>all</td>
<td>Platform to use in the DDL or DML scripts (such as schema-${platform}.sql or data-${platform}.sql).</td>
</tr>
<tr>
<td>spring.datasource.schema</td>
<td></td>
<td>Schema (DDL) script resource references.</td>
</tr>
<tr>
<td>spring.datasource.schema-password</td>
<td></td>
<td>Password of the database to execute DDL scripts (if different).</td>
</tr>
<tr>
<td>spring.datasource.schema-username</td>
<td></td>
<td>Username of the database to execute DDL scripts (if different).</td>
</tr>
<tr>
<td>spring.datasource.separator</td>
<td>;</td>
<td>Statement separator in SQL initialization scripts.</td>
</tr>
<tr>
<td>spring.datasource.sql-script-encoding</td>
<td></td>
<td>SQL scripts encoding.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.datasource.tomcat.abandon-when-percentage-full</td>
<td></td>
<td>Tomcat datasource specific settings bound to an instance of Tomcat JDBC's DataSource</td>
</tr>
<tr>
<td>spring.datasource.tomcat.access-to-underlying-connection-allowed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.alternate-username-allowed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.commit-on-return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.connection-properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.data-source</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.data-source-j-n-d-i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.db-properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.default-auto-commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.default-catalog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.default-transaction-isolation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.driver-class-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.fair-queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.ignore-exception-on-pre-load</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.init-s-q-l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.initial-size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.jdbc-interceptors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.jmx-enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.log-abandoned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.log-validation-errors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.login-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.max-active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.max-age</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.max-idle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.max-wait</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.min-evictable-idle-time-millis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.min-idle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.num-tests-per-eviction-run</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.password</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.propagate-interrupt-state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.remove-abandoned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.remove-abandoned-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.rollback-on-return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.suspect-timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.test-on-borrow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.test-on-connect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.datasource.tomcat.test-on-return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.datasource.type</td>
<td></td>
<td>Fully qualified name of the connection pool implementation to use. By default, it is auto-detected from the classpath.</td>
</tr>
<tr>
<td>spring.datasource.url</td>
<td></td>
<td>JDBC URL of the database.</td>
</tr>
<tr>
<td>spring.datasource.username</td>
<td></td>
<td>Login username of the database.</td>
</tr>
<tr>
<td>spring.datasource.xa.data-source-class-name</td>
<td></td>
<td>XA datasource fully qualified name.</td>
</tr>
<tr>
<td>spring.datasource.xa.properties.*</td>
<td></td>
<td>Properties to pass to the XA data source.</td>
</tr>
<tr>
<td>spring.elasticsearch.rest.connection-timeout</td>
<td>1s</td>
<td>Connection timeout.</td>
</tr>
<tr>
<td>spring.elasticsearch.rest.password</td>
<td></td>
<td>Credentials password.</td>
</tr>
<tr>
<td>spring.elasticsearch.rest.read-timeout</td>
<td>30s</td>
<td>Read timeout.</td>
</tr>
<tr>
<td>spring.elasticsearch.rest.username</td>
<td></td>
<td>Credentials username.</td>
</tr>
<tr>
<td>spring.h2.console.enabled</td>
<td>false</td>
<td>Whether to enable the console.</td>
</tr>
<tr>
<td>spring.h2.console.path</td>
<td>/h2-console</td>
<td>Path at which the console is available.</td>
</tr>
<tr>
<td>spring.h2.console.settings.trace</td>
<td>false</td>
<td>Whether to enable trace output.</td>
</tr>
<tr>
<td>spring.h2.console.settings.web-admin-password</td>
<td></td>
<td>Password to access preferences and tools of H2 Console.</td>
</tr>
<tr>
<td>spring.h2.console.settings.web-allow-others</td>
<td>false</td>
<td>Whether to enable remote access.</td>
</tr>
<tr>
<td>spring.influx.password</td>
<td></td>
<td>Login password.</td>
</tr>
<tr>
<td>spring.influx.url</td>
<td></td>
<td>URL of the InfluxDB instance to which to connect.</td>
</tr>
<tr>
<td>spring.influx.user</td>
<td></td>
<td>Login user.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.jdbc.template.fetch-size</code></td>
<td><code>-1.0</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>spring.jdbc.template.max-rows</code></td>
<td><code>-1.0</code></td>
<td>Maximum number of rows. Use -1 to use the JDBC driver's default configuration.</td>
</tr>
<tr>
<td><code>spring.jdbc.template.query-timeout</code></td>
<td></td>
<td>Query timeout. Default is to use the JDBC driver's default configuration. If a duration suffix is not specified, seconds will be used.</td>
</tr>
<tr>
<td><code>spring.jooq.sql-dialect</code></td>
<td></td>
<td>SQL dialect to use. Auto-detected by default.</td>
</tr>
<tr>
<td><code>spring.jpa.database</code></td>
<td></td>
<td>Target database to operate on, auto-detected by default. Can be alternatively set using the &quot;databasePlatform&quot; property.</td>
</tr>
<tr>
<td><code>spring.jpa.database-platform</code></td>
<td></td>
<td>Name of the target database to operate on, auto-detected by default. Can be alternatively set using the &quot;Database&quot; enum.</td>
</tr>
<tr>
<td><code>spring.jpa.generate-ddl</code></td>
<td><code>false</code></td>
<td>Whether to initialize the schema on startup.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>spring.jpa.hibernate.ddl-auto</td>
<td></td>
<td>DDL mode. This is actually a shortcut for the &quot;hibernate.hbm2ddl.auto&quot; property. Defaults to &quot;create-drop&quot; when using an embedded database and no schema manager was detected. Otherwise, defaults to &quot;none&quot;.</td>
</tr>
<tr>
<td>spring.jpa.hibernate.naming.implicit-strategy</td>
<td></td>
<td>Fully qualified name of the implicit naming strategy.</td>
</tr>
<tr>
<td>spring.jpa.hibernate.naming.physical-strategy</td>
<td></td>
<td>Fully qualified name of the physical naming strategy.</td>
</tr>
<tr>
<td>spring.jpa.hibernate.use-new-id-generator-mappings</td>
<td></td>
<td>Whether to use Hibernate's newer IdentifierGenerator for AUTO, TABLE and SEQUENCE. This is actually a shortcut for the &quot;hibernate.id.new_generator_mappings&quot; property. When not specified will default to &quot;true&quot;.</td>
</tr>
<tr>
<td>spring.jpa.mapping-resources</td>
<td></td>
<td>Mapping resources (equivalent to &quot;mapping-file&quot; entries in persistence.xml).</td>
</tr>
<tr>
<td>spring.jpa.open-in-view</td>
<td>true</td>
<td>Register OpenEntityManagerInViewInterceptor. Binds a JPA EntityManager to the thread for the entire processing of the request.</td>
</tr>
<tr>
<td>spring.jpa.properties.*</td>
<td></td>
<td>Additional native properties to set on the JPA provider.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jpa.show-sql</td>
<td>false</td>
<td>Whether to enable logging of SQL statements.</td>
</tr>
<tr>
<td>spring.ldap.anonymous-read-only</td>
<td></td>
<td>Whether read-only operations should use an anonymous environment. Disabled by default unless a username is set.</td>
</tr>
<tr>
<td>spring.ldap.base</td>
<td></td>
<td>Base suffix from which all operations should originate.</td>
</tr>
<tr>
<td>spring.ldap.base-environment.*</td>
<td></td>
<td>LDAP specification settings.</td>
</tr>
<tr>
<td>spring.ldap.embedded.base-dn</td>
<td></td>
<td>List of base DNs.</td>
</tr>
<tr>
<td>spring.ldap.embedded.credential.password</td>
<td></td>
<td>Embedded LDAP password.</td>
</tr>
<tr>
<td>spring.ldap.embedded.credential.username</td>
<td></td>
<td>Embedded LDAP username.</td>
</tr>
<tr>
<td>spring.ldap.embedded.ldif</td>
<td>classpath: schema.ldif</td>
<td>Schema (LDIF) script resource reference.</td>
</tr>
<tr>
<td>spring.ldap.embedded.port</td>
<td>0.0</td>
<td>Embedded LDAP port.</td>
</tr>
<tr>
<td>spring.ldap.embedded.validation.enabled</td>
<td>true</td>
<td>Whether to enable LDAP schema validation.</td>
</tr>
<tr>
<td>spring.ldap.embedded.validation.schema</td>
<td></td>
<td>Path to the custom schema.</td>
</tr>
<tr>
<td>spring.ldap.password</td>
<td></td>
<td>Login password of the server.</td>
</tr>
<tr>
<td>spring.ldap.template.ignore-name-not-found-exception</td>
<td>false</td>
<td>Whether NameNotFoundException should be ignored in searches via the LdapTemplate.</td>
</tr>
<tr>
<td>spring.ldap.template.ignore-partial-result-exception</td>
<td>false</td>
<td>Whether PartialResultException should be ignored in searches via the LdapTemplate.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>spring.ldap.template.ignore-size-limit-exceeded-exception</td>
<td>true</td>
<td>Whether SizeLimitExceededException should be ignored in searches via the LdapTemplate.</td>
</tr>
<tr>
<td>spring.ldap.urls</td>
<td></td>
<td>LDAP URLs of the server.</td>
</tr>
<tr>
<td>spring.ldap.username</td>
<td></td>
<td>Login username of the server.</td>
</tr>
<tr>
<td>spring.mongodb.embedded.features</td>
<td>[sync_delay]</td>
<td>Comma-separated list of features to enable. Uses the defaults of the configured version by default.</td>
</tr>
<tr>
<td>spring.mongodb.embedded.storage.database-dir</td>
<td></td>
<td>Directory used for data storage.</td>
</tr>
<tr>
<td>spring.mongodb.embedded.storage.oplog-size</td>
<td></td>
<td>Maximum size of the oplog.</td>
</tr>
<tr>
<td>spring.mongodb.embedded.storage.repl-set-name</td>
<td></td>
<td>Name of the replica set.</td>
</tr>
<tr>
<td>spring.mongodb.embedded.version</td>
<td>3.5.5</td>
<td>Version of Mongo to use.</td>
</tr>
<tr>
<td>spring.neo4j.authentication.kerberos-ticket</td>
<td></td>
<td>Kerberos ticket for connecting to the database. Mutual exclusive with a given username.</td>
</tr>
<tr>
<td>spring.neo4j.authentication.password</td>
<td></td>
<td>Login password of the server.</td>
</tr>
<tr>
<td>spring.neo4j.authentication.realm</td>
<td></td>
<td>Realm to connect to.</td>
</tr>
<tr>
<td>spring.neo4j.authentication.username</td>
<td></td>
<td>Login user of the server.</td>
</tr>
<tr>
<td>spring.neo4j.connection-timeout</td>
<td>30s</td>
<td>Timeout for borrowing connections from the pool.</td>
</tr>
<tr>
<td>spring.neo4j.max-transaction-retry-time</td>
<td>30s</td>
<td>Maximum time transactions are allowed to retry.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.neo4j.pool.connection-acquisition-timeout</td>
<td>60s</td>
<td>Acquisition of new connections will be attempted for at most configured timeout.</td>
</tr>
<tr>
<td>spring.neo4j.pool.idle-time-before-connection-test</td>
<td></td>
<td>Pooled connections that have been idle in the pool for longer than this threshold will be tested before they are used again.</td>
</tr>
<tr>
<td>spring.neo4j.pool.log-leaked-sessions</td>
<td>false</td>
<td>Whether to log leaked sessions.</td>
</tr>
<tr>
<td>spring.neo4j.pool.max-connection-lifetime</td>
<td>1h</td>
<td>Pooled connections older than this threshold will be closed and removed from the pool.</td>
</tr>
<tr>
<td>spring.neo4j.pool.max-connection-pool-size</td>
<td>100.0</td>
<td>Maximum amount of connections in the connection pool towards a single database.</td>
</tr>
<tr>
<td>spring.neo4j.pool.metrics-enabled</td>
<td>false</td>
<td>Whether to enable metrics.</td>
</tr>
<tr>
<td>spring.neo4j.security.cert-file</td>
<td></td>
<td>Path to the file that holds the trusted certificates.</td>
</tr>
<tr>
<td>spring.neo4j.security.encrypted</td>
<td>false</td>
<td>Whether the driver should use encrypted traffic.</td>
</tr>
<tr>
<td>spring.neo4j.security.hostname-verification-enabled</td>
<td>true</td>
<td>Whether hostname verification is required.</td>
</tr>
<tr>
<td>spring.neo4j.security.trust-strategy</td>
<td>trust-system-ca-signed-certificates</td>
<td>Trust strategy to use.</td>
</tr>
<tr>
<td>spring.neo4j.uri</td>
<td>bolt://localhost:7687</td>
<td>URI used by the driver.</td>
</tr>
<tr>
<td>spring.r2dbc.generate-unique-name</td>
<td>false</td>
<td>Whether to generate a random database name. Ignore any configured name when enabled.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.r2dbc.name</td>
<td></td>
<td>Database name. Set if no name is specified in the url. Default to “testdb” when using an embedded database.</td>
</tr>
<tr>
<td>spring.r2dbc.password</td>
<td></td>
<td>Login password of the database. Set if no password is specified in the url.</td>
</tr>
<tr>
<td>spring.r2dbc.pool.enabled</td>
<td></td>
<td>Whether pooling is enabled. Enabled automatically if “r2dbc-pool” is on the classpath.</td>
</tr>
<tr>
<td>spring.r2dbc.pool.initial-size</td>
<td>10.0</td>
<td>Initial connection pool size.</td>
</tr>
<tr>
<td>spring.r2dbc.pool.max-acquire-time</td>
<td></td>
<td>Maximum time to acquire a connection from the pool. By default, wait indefinitely.</td>
</tr>
<tr>
<td>spring.r2dbc.pool.max-create-connection-time</td>
<td></td>
<td>Maximum time to wait to create a new connection. By default, wait indefinitely.</td>
</tr>
<tr>
<td>spring.r2dbc.pool.max-idle-time</td>
<td>30m</td>
<td>Maximum amount of time that a connection is allowed to sit idle in the pool.</td>
</tr>
<tr>
<td>spring.r2dbc.pool.max-life-time</td>
<td></td>
<td>Maximum lifetime of a connection in the pool. By default, connections have an infinite lifetime.</td>
</tr>
<tr>
<td>spring.r2dbc.pool.max-size</td>
<td>10.0</td>
<td>Maximal connection pool size.</td>
</tr>
<tr>
<td>spring.r2dbc.pool.validation-depth</td>
<td>local</td>
<td>Validation depth.</td>
</tr>
<tr>
<td>spring.r2dbc.pool.validation-query</td>
<td></td>
<td>Validation query.</td>
</tr>
<tr>
<td>spring.r2dbc.properties.*</td>
<td></td>
<td>Additional R2DBC options.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.r2dbc.url</td>
<td></td>
<td>R2DBC URL of the database. database name, username, password and pooling options specified in the url take precedence over individual options.</td>
</tr>
<tr>
<td>spring.r2dbc.username</td>
<td></td>
<td>Login username of the database. Set if no username is specified in the url.</td>
</tr>
<tr>
<td>spring.redis.client-name</td>
<td></td>
<td>Client name to be set on connections with CLIENT SETNAME.</td>
</tr>
<tr>
<td>spring.redis.client-type</td>
<td></td>
<td>Type of client to use. By default, auto-detected according to the classpath.</td>
</tr>
<tr>
<td>spring.redis.cluster.max-redirects</td>
<td></td>
<td>Maximum number of redirects to follow when executing commands across the cluster.</td>
</tr>
<tr>
<td>spring.redis.cluster.nodes</td>
<td></td>
<td>Comma-separated list of &quot;host:port&quot; pairs to bootstrap from. This represents an &quot;initial&quot; list of cluster nodes and is required to have at least one entry.</td>
</tr>
<tr>
<td>spring.redis.connect-timeout</td>
<td></td>
<td>Connection timeout.</td>
</tr>
<tr>
<td>spring.redis.database</td>
<td>0.0</td>
<td>Database index used by the connection factory.</td>
</tr>
<tr>
<td>spring.redis.host</td>
<td>localhost</td>
<td>Redis server host.</td>
</tr>
<tr>
<td>spring.redis.jedis.pool.max-active</td>
<td>8.0</td>
<td>Maximum number of connections that can be allocated by the pool at a given time. Use a negative value for no limit.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.redis.jedis.pool.max-idle</td>
<td>8.0</td>
<td>Maximum number of &quot;idle&quot; connections in the pool. Use a negative value to indicate an unlimited number of idle connections.</td>
</tr>
<tr>
<td>spring.redis.jedis.pool.max-wait</td>
<td>-1ms</td>
<td>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.</td>
</tr>
<tr>
<td>spring.redis.jedis.pool.min-idle</td>
<td>0.0</td>
<td>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.</td>
</tr>
<tr>
<td>spring.redis.jedis.pool.time-between-eviction-runs</td>
<td></td>
<td>Time between runs of the idle object evictor thread. When positive, the idle object evictor thread starts, otherwise no idle object eviction is performed.</td>
</tr>
<tr>
<td>spring.redis.lettuce.cluster.refresh.adaptive</td>
<td>false</td>
<td>Whether adaptive topology refreshing using all available refresh triggers should be used.</td>
</tr>
<tr>
<td>spring.redis.lettuce.cluster.refresh.dynamic-refresh-sources</td>
<td>true</td>
<td>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.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.redis.lettuce.cluster.refresh.period</code></td>
<td></td>
<td>Cluster topology refresh period.</td>
</tr>
<tr>
<td><code>spring.redis.lettuce.pool.max-active</code></td>
<td>8.0</td>
<td>Maximum number of connections that can be allocated by the pool at a given time. Use a negative value for no limit.</td>
</tr>
<tr>
<td><code>spring.redis.lettuce.pool.max-idle</code></td>
<td>8.0</td>
<td>Maximum number of &quot;idle&quot; connections in the pool. Use a negative value to indicate an unlimited number of idle connections.</td>
</tr>
<tr>
<td><code>spring.redis.lettuce.pool.max-wait</code></td>
<td>-1ms</td>
<td>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.</td>
</tr>
<tr>
<td><code>spring.redis.lettuce.pool.min-idle</code></td>
<td>0.0</td>
<td>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.</td>
</tr>
<tr>
<td><code>spring.redis.lettuce.pool.time-between-eviction-runs</code></td>
<td></td>
<td>Time between runs of the idle object evictor thread. When positive, the idle object evictor thread starts, otherwise no idle object eviction is performed.</td>
</tr>
<tr>
<td><code>spring.redis.lettuce.shutdown-timeout</code></td>
<td>100ms</td>
<td>Shutdown timeout.</td>
</tr>
<tr>
<td><code>spring.redis.password</code></td>
<td></td>
<td>Login password of the redis server.</td>
</tr>
<tr>
<td><code>spring.redis.port</code></td>
<td>6379.0</td>
<td>Redis server port.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>spring.redis.sentinel.master</td>
<td></td>
<td>Name of the Redis server.</td>
</tr>
<tr>
<td>spring.redis.sentinel.nodes</td>
<td></td>
<td>Comma-separated list of &quot;host:port&quot; pairs.</td>
</tr>
<tr>
<td>spring.redis.sentinel.password</td>
<td></td>
<td>Password for authenticating with sentinel(s).</td>
</tr>
<tr>
<td>spring.redis.ssh</td>
<td>false</td>
<td>Whether to enable SSL support.</td>
</tr>
<tr>
<td>spring.redis.timeout</td>
<td></td>
<td>Read timeout.</td>
</tr>
<tr>
<td>spring.redis.url</td>
<td></td>
<td>Connection URL. Overrides host, port, and password. User is ignored. Example: redis://user:<a href="mailto:password@example.com">password@example.com</a>:6379</td>
</tr>
<tr>
<td>spring.redis.username</td>
<td></td>
<td>Login username of the redis server.</td>
</tr>
</tbody>
</table>

### 10.A.6. Transaction Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.jta.atomikos.connectionfactory.borrow-connection-timeout</td>
<td>30.0</td>
<td>Timeout, in seconds, for borrowing connections from the pool.</td>
</tr>
<tr>
<td>spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag</td>
<td>true</td>
<td>Whether to ignore the transacted flag when creating session.</td>
</tr>
<tr>
<td>spring.jta.atomikos.connectionfactory.local-transaction-mode</td>
<td>false</td>
<td>Whether local transactions are desired.</td>
</tr>
<tr>
<td>spring.jta.atomikos.connectionfactory.maintenance-interval</td>
<td>60.0</td>
<td>Time, in seconds, between runs of the pool's maintenance thread.</td>
</tr>
<tr>
<td>spring.jta.atomikos.connectionfactory.max-idle-time</td>
<td>60.0</td>
<td>Time, in seconds, after which connections are cleaned up from the pool.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jta.atomikos.connectionfactory.max-lifetime</td>
<td>0.0</td>
<td>Time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>Maximum size of the pool.</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>Minimum size of the pool.</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>Reap timeout, in seconds, for borrowed connections. 0 denotes no limit.</td>
</tr>
<tr>
<td></td>
<td>jmsConnectionFactory</td>
<td>Unique name used to identify the resource during recovery.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vendor-specific implementation of XAConnectionFactory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vendor-specific XA properties.</td>
</tr>
<tr>
<td></td>
<td>30.0</td>
<td>Timeout, in seconds, for borrowing connections from the pool.</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>Whether to use concurrent connection validation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default isolation level of connections provided by the pool.</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>Timeout, in seconds, for establishing a database connection.</td>
</tr>
<tr>
<td></td>
<td>60.0</td>
<td>Time, in seconds, between runs of the pool's maintenance thread.</td>
</tr>
<tr>
<td></td>
<td>60.0</td>
<td>Time, in seconds, after which connections are cleaned up from the pool.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jta.atomikos.datasource.max-lifetime</td>
<td>0.0</td>
<td>Time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.</td>
</tr>
<tr>
<td>spring.jta.atomikos.datasource.max-pool-size</td>
<td>1.0</td>
<td>Maximum size of the pool.</td>
</tr>
<tr>
<td>spring.jta.atomikos.datasource.min-pool-size</td>
<td>1.0</td>
<td>Minimum size of the pool.</td>
</tr>
<tr>
<td>spring.jta.atomikos.datasource.reap-timeout</td>
<td>0.0</td>
<td>Reap timeout, in seconds, for borrowed connections. 0 denotes no limit.</td>
</tr>
<tr>
<td>spring.jta.atomikos.datasource.test-query</td>
<td></td>
<td>SQL query or statement used to validate a connection before returning it.</td>
</tr>
<tr>
<td>spring.jta.atomikos.datasource.unique-resource-name</td>
<td>dataSource</td>
<td>Unique name used to identify the resource during recovery.</td>
</tr>
<tr>
<td>spring.jta.atomikos.datasource.xa-data-source-class-name</td>
<td></td>
<td>Vendor-specific implementation of XAConnectionFactory.</td>
</tr>
<tr>
<td>spring.jta.atomikos.datasource.xa-properties</td>
<td></td>
<td>Vendor-specific XA properties.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.allow-sub-transactions</td>
<td>true</td>
<td>Specify whether sub-transactions are allowed.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.checkpoint-interval</td>
<td>500.0</td>
<td>Interval between checkpoints, expressed as the number of log writes between two checkpoints. A checkpoint reduces the log file size at the expense of adding some overhead in the runtime.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.default-jta-timeout</td>
<td>10000ms</td>
<td>Default timeout for JTA transactions.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.default-max-wait-time-on-shutdown</td>
<td></td>
<td>How long should normal shutdown (no-force) wait for transactions to complete.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.enable-logging</td>
<td>true</td>
<td>Whether to enable disk logging.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.force-shutdown-on-vm-exit</td>
<td>false</td>
<td>Whether a VM shutdown should trigger forced shutdown of the transaction core.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.log-base-dir</td>
<td></td>
<td>Directory in which the log files should be stored. Defaults to the current working directory.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.log-base-name</td>
<td>tmlog</td>
<td>Transactions log file base name.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.max-actives</td>
<td>50.0</td>
<td>Maximum number of active transactions.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.max-timeout</td>
<td>300000ms</td>
<td>Maximum timeout that can be allowed for transactions.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.recovery.delay</td>
<td>10000ms</td>
<td>Delay between two recovery scans.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.recovery.forget-orphaned-log-entries-delay</td>
<td>86400000ms</td>
<td>Delay after which recovery can cleanup pending ('orphaned') log entries.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.recovery.max-retries</td>
<td>5.0</td>
<td>Number of retry attempts to commit the transaction before throwing an exception.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.recovery.retry-interval</td>
<td>10000ms</td>
<td>Delay between retry attempts.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.serial-jta-transactions</td>
<td>true</td>
<td>Whether subtransactions should be joined when possible.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.service</td>
<td></td>
<td>Transaction manager implementation that should be started.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.threaded-two-phase-commit</td>
<td>false</td>
<td>Whether to use different (and concurrent) threads for two-phase commit on the participating resources.</td>
</tr>
<tr>
<td>spring.jta.atomikos.properties.transaction-manager-unique-name</td>
<td></td>
<td>The transaction manager's unique name. Defaults to the machine's IP address. If you plan to run more than one transaction manager against one database you must set this property to a unique value.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.acquire-increment</td>
<td>1.0</td>
<td>Number of connections to create when growing the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.acquisition-interval</td>
<td>1.0</td>
<td>Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was acquired.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.acquisition-timeout</td>
<td>30.0</td>
<td>Timeout, in seconds, for acquiring connections from the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.allow-local-transactions</td>
<td>false</td>
<td>Whether the transaction manager should allow mixing XA and non-XA transactions.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.apply-transaction-timeout</td>
<td>false</td>
<td>Whether the transaction timeout should be set on the XAResource when it is enlisted.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.automatic-enlisting-enabled</td>
<td>true</td>
<td>Whether resources should be enlisted and delisted automatically.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.cache-producers-consumers</td>
<td>true</td>
<td>Whether producers and consumers should be cached.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.class-name</td>
<td></td>
<td>Underlying implementation class name of the XA resource.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.defer-connection-release</td>
<td>true</td>
<td>Whether the provider can run many transactions on the same connection and supports transaction interleaving.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.disabled</td>
<td>false</td>
<td>Whether this resource is disabled, meaning it's temporarily forbidden to acquire a connection from its pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.driver-properties</td>
<td></td>
<td>Properties that should be set on the underlying implementation.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.failed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.ignore-recovery-failures</td>
<td>false</td>
<td>Whether recovery failures should be ignored.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.max-idle-time</td>
<td>60.0</td>
<td>Time, in seconds, after which connections are cleaned up from the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.max-pool-size</td>
<td>0.0</td>
<td>Maximum size of the pool. 0 denotes no limit.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.min-pool-size</td>
<td>0.0</td>
<td>Minimum size of the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.password</td>
<td></td>
<td>Password to use to connect to the JMS provider.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.share-transaction-connections</td>
<td>false</td>
<td>Whether connections in the ACCESSIBLE state can be shared within the context of a transaction.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.test-connections</td>
<td>false</td>
<td>Whether connections should be tested when acquired from the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.two-pc-ordering-position</td>
<td>1.0</td>
<td>Position that this resource should take during two-phase commit (always first is Integer.MIN_VALUE, always last is Integer.MAX_VALUE).</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.unique-name</td>
<td>jmsConnectionFactory</td>
<td>Unique name used to identify the resource during recovery.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.use-tm-join</td>
<td>true</td>
<td>Whether TMJOIN should be used when starting XAResources.</td>
</tr>
<tr>
<td>spring.jta.bitronix.connectionfactory.user</td>
<td></td>
<td>User to use to connect to the JMS provider.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.acquire-increment</td>
<td>1.0</td>
<td>Number of connections to create when growing the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.acquisition-interval</td>
<td>1.0</td>
<td>Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was acquired.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.acquisition-timeout</td>
<td>30.0</td>
<td>Timeout, in seconds, for acquiring connections from the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.allow-local-transactions</td>
<td>false</td>
<td>Whether the transaction manager should allow mixing XA and non-XA transactions.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.apply-transaction-timeout</td>
<td>false</td>
<td>Whether the transaction timeout should be set on the XAResource when it is enlisted.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.automatic-enlisting-enabled</td>
<td>true</td>
<td>Whether resources should be enlisted and delisted automatically.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.class-name</td>
<td></td>
<td>Underlying implementation class name of the XA resource.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.cursor-holdability</td>
<td></td>
<td>Default cursor holdability for connections.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.defer-connection-release</td>
<td>true</td>
<td>Whether the database can run many transactions on the same connection and supports transaction interleaving.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.disabled</td>
<td>false</td>
<td>Whether this resource is disabled, meaning it’s temporarily forbidden to acquire a connection from its pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.driver-properties</td>
<td></td>
<td>Properties that should be set on the underlying implementation.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.enable-jdbc4-connection-test</td>
<td>false</td>
<td>Whether Connection.isValid() is called when acquiring a connection from the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.failed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.ignore-recovery-failures</td>
<td>false</td>
<td>Whether recovery failures should be ignored.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.isolation-level</td>
<td></td>
<td>Default isolation level for connections.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.local-auto-commit</td>
<td></td>
<td>Default auto-commit mode for local transactions.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.login-timeout</td>
<td></td>
<td>Timeout, in seconds, for establishing a database connection.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.max-idle-time</td>
<td>60.0</td>
<td>Time, in seconds, after which connections are cleaned up from the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.max-pool-size</td>
<td>0.0</td>
<td>Maximum size of the pool. 0 denotes no limit.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.min-pool-size</td>
<td>0.0</td>
<td>Minimum size of the pool.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.prepared-statement-cache-size</td>
<td>0.0</td>
<td>Target size of the prepared statement cache. 0 disables the cache.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.share-transaction-connections</td>
<td>false</td>
<td>Whether connections in the ACCESSIBLE state can be shared within the context of a transaction.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.test-query</td>
<td></td>
<td>SQL query or statement used to validate a connection before returning it.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.two-pc-ordering-position</td>
<td>1.0</td>
<td>Position that this resource should take during two-phase commit (always first is Integer.MIN_VALUE, and always last is Integer.MAX_VALUE).</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.unique-name</td>
<td>dataSource</td>
<td>Unique name used to identify the resource during recovery.</td>
</tr>
<tr>
<td>spring.jta.bitronix.datasource.use-tm-join</td>
<td>true</td>
<td>Whether TMJOIN should be used when starting XAResources.</td>
</tr>
<tr>
<td>spring.jta.enabled</td>
<td>true</td>
<td>Whether to enable JTA support.</td>
</tr>
</tbody>
</table>
### Key

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.jta.log-dir</code></td>
<td></td>
<td>Transaction logs directory.</td>
</tr>
<tr>
<td><code>spring.jta.transaction-manager-id</code></td>
<td></td>
<td>Transaction manager unique identifier.</td>
</tr>
<tr>
<td><code>spring.transaction.default-timeout</code></td>
<td></td>
<td>Default transaction timeout. If a duration suffix is not specified, seconds will be used.</td>
</tr>
<tr>
<td><code>spring.transaction.rollback-on-commit-failure</code></td>
<td></td>
<td>Whether to roll back on commit failures.</td>
</tr>
</tbody>
</table>

### 10.A.7. Data Migration Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.flyway.baseline-description</code></td>
<td><code>&lt;&lt; Flyway Baseline &gt;&gt;</code></td>
<td>Description to tag an existing schema with when applying a baseline.</td>
</tr>
<tr>
<td><code>spring.flyway.baseline-on-migrate</code></td>
<td><code>false</code></td>
<td>Whether to automatically call baseline when migrating a non-empty schema.</td>
</tr>
<tr>
<td><code>spring.flyway.baseline-version</code></td>
<td><code>1</code></td>
<td>Version to tag an existing schema with when executing baseline.</td>
</tr>
<tr>
<td><code>spring.flyway.batch</code></td>
<td></td>
<td>Whether to batch SQL statements when executing them. Requires Flyway Teams.</td>
</tr>
<tr>
<td><code>spring.flyway.check-location</code></td>
<td><code>true</code></td>
<td>Whether to check that migration scripts location exists.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.flyway.cherry-pick</td>
<td></td>
<td>Migrations that Flyway should consider when migrating or undoing. When empty all available migrations are considered. Requires Flyway Teams.</td>
</tr>
<tr>
<td>spring.flyway.clean-disabled</td>
<td>false</td>
<td>Whether to disable cleaning of the database.</td>
</tr>
<tr>
<td>spring.flyway.clean-on-validation-error</td>
<td>false</td>
<td>Whether to automatically call clean when a validation error occurs.</td>
</tr>
<tr>
<td>spring.flyway.connect-retries</td>
<td>0.0</td>
<td>Maximum number of retries when attempting to connect to the database.</td>
</tr>
<tr>
<td>spring.flyway.create-schemas</td>
<td>true</td>
<td>Whether Flyway should attempt to create the schemas specified in the schemas property.</td>
</tr>
<tr>
<td>spring.flyway.default-schema</td>
<td></td>
<td>Default schema name managed by Flyway (case-sensitive).</td>
</tr>
<tr>
<td>spring.flyway.enabled</td>
<td>true</td>
<td>Whether to enable flyway.</td>
</tr>
<tr>
<td>spring.flyway.encoding</td>
<td>UTF-8</td>
<td>Encoding of SQL migrations.</td>
</tr>
<tr>
<td>spring.flyway.error-overrides</td>
<td></td>
<td>Rules for the built-in error handling to override specific SQL states and error codes. Requires Flyway Teams.</td>
</tr>
<tr>
<td>spring.flyway.group</td>
<td>false</td>
<td>Whether to group all pending migrations together in the same transaction when applying them.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.flyway.ignore-future-migrations</td>
<td>true</td>
<td>Whether to ignore future migrations when reading the schema history table.</td>
</tr>
<tr>
<td>spring.flyway.ignore-ignored-migrations</td>
<td>false</td>
<td>Whether to ignore ignored migrations when reading the schema history table.</td>
</tr>
<tr>
<td>spring.flyway.ignore-missing-migrations</td>
<td>false</td>
<td>Whether to ignore missing migrations when reading the schema history table.</td>
</tr>
<tr>
<td>spring.flyway.ignore-pending-migrations</td>
<td>false</td>
<td>Whether to ignore pending migrations when reading the schema history table.</td>
</tr>
<tr>
<td>spring.flyway.init-sqls</td>
<td></td>
<td>SQL statements to execute to initialize a connection immediately after obtaining it.</td>
</tr>
<tr>
<td>spring.flyway.installed-by</td>
<td></td>
<td>Username recorded in the schema history table as having applied the migration.</td>
</tr>
<tr>
<td>spring.flyway.jdbc-properties.*</td>
<td></td>
<td>Properties to pass to the JDBC driver. Requires Flyway Teams.</td>
</tr>
<tr>
<td>spring.flyway.license-key</td>
<td></td>
<td>Licence key for Flyway Teams.</td>
</tr>
<tr>
<td>spring.flyway.locations</td>
<td>[classpath:db/migration]</td>
<td>Locations of migrations scripts. Can contain the special &quot;{vendor}&quot; placeholder to use vendor-specific locations.</td>
</tr>
<tr>
<td>spring.flyway.lock-retry-count</td>
<td>50.0</td>
<td>Maximum number of retries when trying to obtain a lock.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.flyway.mixed</td>
<td>false</td>
<td>Whether to allow mixing transactional and non-transactional statements within the same migration.</td>
</tr>
<tr>
<td>spring.flyway.oracle-kerberos-config-file</td>
<td></td>
<td>Path of the Oracle Kerberos config file. Requires Flyway Teams.</td>
</tr>
<tr>
<td>spring.flyway.oracle-sqlplus</td>
<td></td>
<td>Whether to enable support for Oracle SQL*Plus commands. Requires Flyway Teams.</td>
</tr>
<tr>
<td>spring.flyway.oracle-sqlplus-warn</td>
<td></td>
<td>Whether to issue a warning rather than an error when a not-yet-supported Oracle SQL*Plus statement is encountered. Requires Flyway Teams.</td>
</tr>
<tr>
<td>spring.flyway.out-of-order</td>
<td>false</td>
<td>Whether to allow migrations to be run out of order.</td>
</tr>
<tr>
<td>spring.flyway.output-query-results</td>
<td></td>
<td>Whether Flyway should output a table with the results of queries when executing migrations. Requires Flyway Teams.</td>
</tr>
<tr>
<td>spring.flyway.password</td>
<td></td>
<td>Login password of the database to migrate.</td>
</tr>
<tr>
<td>spring.flyway.placeholder-prefix</td>
<td>${</td>
<td>Prefix of placeholders in migration scripts.</td>
</tr>
<tr>
<td>spring.flyway.placeholder-replacement</td>
<td>true</td>
<td>Perform placeholder replacement in migration scripts.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.flyway.placeholder-suffix</td>
<td><code>}</code></td>
<td>Suffix of placeholders in migration scripts.</td>
</tr>
<tr>
<td>spring.flyway.placeholders.*</td>
<td></td>
<td>Placeholders and their replacements to apply to sql migration scripts.</td>
</tr>
<tr>
<td>spring.flyway.repeatable-sql-migration-prefix</td>
<td>R</td>
<td>File name prefix for repeatable SQL migrations.</td>
</tr>
<tr>
<td>spring.flyway.schemas</td>
<td></td>
<td>Scheme names managed by Flyway (case-sensitive).</td>
</tr>
<tr>
<td>spring.flyway.skip-default-callbacks</td>
<td><code>false</code></td>
<td>Whether to skip default callbacks. If true, only custom callbacks are used.</td>
</tr>
<tr>
<td>spring.flyway.skip-default-resolvers</td>
<td><code>false</code></td>
<td>Whether to skip default resolvers. If true, only custom resolvers are used.</td>
</tr>
<tr>
<td>spring.flyway.skip-executing-migrations</td>
<td></td>
<td>Whether Flyway should skip executing the contents of the migrations and only update the schema history table. Requires Flyway teams.</td>
</tr>
<tr>
<td>spring.flyway.sql-migration-prefix</td>
<td>V</td>
<td>File name prefix for SQL migrations.</td>
</tr>
<tr>
<td>spring.flyway.sql-migration-separator</td>
<td><code>__</code></td>
<td>File name separator for SQL migrations.</td>
</tr>
<tr>
<td>spring.flyway.sql-migration-suffixes</td>
<td><code>[.sql]</code></td>
<td>File name suffix for SQL migrations.</td>
</tr>
<tr>
<td>spring.flyway.stream</td>
<td></td>
<td>Whether to stream SQL migrations when executing them. Requires Flyway Teams.</td>
</tr>
<tr>
<td>spring.flyway.table</td>
<td><code>flyway_schema_history</code></td>
<td>Name of the schema history table that will be used by Flyway.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.flyway.tablespace</code></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><code>spring.flyway.target</code></td>
<td></td>
<td>Target version up to which migrations should be considered.</td>
</tr>
<tr>
<td><code>spring.flyway.url</code></td>
<td></td>
<td>JDBC url of the database to migrate. If not set, the primary configured data source is used.</td>
</tr>
<tr>
<td><code>spring.flyway.user</code></td>
<td></td>
<td>Login user of the database to migrate.</td>
</tr>
<tr>
<td><code>spring.flyway.validate-migration-naming</code></td>
<td><code>false</code></td>
<td>Whether to validate migrations and callbacks whose scripts do not obey the correct naming convention.</td>
</tr>
<tr>
<td><code>spring.flyway.validate-on-migrate</code></td>
<td><code>true</code></td>
<td>Whether to automatically call validate when performing a migration.</td>
</tr>
<tr>
<td><code>spring.liquibase.change-log</code></td>
<td></td>
<td>Change log configuration path.</td>
</tr>
<tr>
<td><code>spring.liquibase.clear-checksums</code></td>
<td><code>false</code></td>
<td>Whether to clear all checksums in the current changelog, so they will be recalculated upon the next update.</td>
</tr>
<tr>
<td><code>spring.liquibase.contexts</code></td>
<td></td>
<td>Comma-separated list of runtime contexts to use.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.liquibase.database-change-log-lock-table</td>
<td>DATABASECHANGELOGLOCK</td>
<td>Name of table to use for tracking concurrent Liquibase usage.</td>
</tr>
<tr>
<td>spring.liquibase.database-change-log-table</td>
<td>DATABASECHANGELOG</td>
<td>Name of table to use for tracking change history.</td>
</tr>
<tr>
<td>spring.liquibase.default-schema</td>
<td></td>
<td>Default database schema.</td>
</tr>
<tr>
<td>spring.liquibase.driver-class-name</td>
<td></td>
<td>Fully qualified name of the JDBC driver. Auto-detected based on the URL by default.</td>
</tr>
<tr>
<td>spring.liquibase.drop-first</td>
<td>false</td>
<td>Whether to first drop the database schema.</td>
</tr>
<tr>
<td>spring.liquibase.enabled</td>
<td>true</td>
<td>Whether to enable Liquibase support.</td>
</tr>
<tr>
<td>spring.liquibase.labels</td>
<td></td>
<td>Comma-separated list of runtime labels to use.</td>
</tr>
<tr>
<td>spring.liquibase.liquibase-schema</td>
<td></td>
<td>Schema to use for Liquibase objects.</td>
</tr>
<tr>
<td>spring.liquibase.liquibase-tablespace</td>
<td></td>
<td>Tablespace to use for Liquibase objects.</td>
</tr>
<tr>
<td>spring.liquibase.parameters.*</td>
<td></td>
<td>Change log parameters.</td>
</tr>
<tr>
<td>spring.liquibase.password</td>
<td></td>
<td>Login password of the database to migrate.</td>
</tr>
<tr>
<td>spring.liquibase.rollback-file</td>
<td></td>
<td>File to which rollback SQL is written when an update is performed.</td>
</tr>
<tr>
<td>spring.liquibase.tag</td>
<td></td>
<td>Tag name to use when applying database changes. Can also be used with &quot;rollbackFile&quot; to generate a rollback script for all existing changes associated with that tag.</td>
</tr>
<tr>
<td>spring.liquibase.test-rollback-on-update</td>
<td>false</td>
<td>Whether rollback should be tested before update is performed.</td>
</tr>
</tbody>
</table>
### 10.A.8. Integration Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.liquibase.url</code></td>
<td></td>
<td>JDBC URL of the database to migrate. If not set, the primary configured data source is used.</td>
</tr>
<tr>
<td><code>spring.liquibase.user</code></td>
<td></td>
<td>Login user of the database to migrate.</td>
</tr>
<tr>
<td><code>spring.activemq.broker-url</code></td>
<td></td>
<td>URL of the ActiveMQ broker. Auto-generated by default.</td>
</tr>
<tr>
<td><code>spring.activemq.close-timeout</code></td>
<td>15s</td>
<td>Time to wait before considering a close complete.</td>
</tr>
<tr>
<td><code>spring.activemq.in-memory</code></td>
<td>true</td>
<td>Whether the default broker URL should be in memory. Ignored if an explicit broker has been specified.</td>
</tr>
<tr>
<td><code>spring.activemq.non-blocking-redelivery</code></td>
<td>false</td>
<td>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.</td>
</tr>
<tr>
<td><code>spring.activemq.packages.trust-all</code></td>
<td></td>
<td>Whether to trust all packages.</td>
</tr>
<tr>
<td><code>spring.activemq.packages.trusted</code></td>
<td></td>
<td>Comma-separated list of specific packages to trust (when not trusting all packages).</td>
</tr>
<tr>
<td><code>spring.activemq.password</code></td>
<td></td>
<td>Login password of the broker.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.activemq.pool.block-if-full</td>
<td>true</td>
<td>Whether to block when a connection is requested and the pool is full. Set it to false to throw a &quot;JMSException&quot; instead.</td>
</tr>
<tr>
<td>spring.activemq.pool.block-if-full-timeout</td>
<td>-1ms</td>
<td>Blocking period before throwing an exception if the pool is still full.</td>
</tr>
<tr>
<td>spring.activemq.pool.enabled</td>
<td>false</td>
<td>Whether a JmsPoolConnectionFactory should be created, instead of a regular ConnectionFactory.</td>
</tr>
<tr>
<td>spring.activemq.pool.idle-timeout</td>
<td>30s</td>
<td>Connection idle timeout.</td>
</tr>
<tr>
<td>spring.activemq.pool.max-connections</td>
<td>1.0</td>
<td>Maximum number of pooled connections.</td>
</tr>
<tr>
<td>spring.activemq.pool.max-sessions-per-connection</td>
<td>500.0</td>
<td>Maximum number of pooled sessions per connection in the pool.</td>
</tr>
<tr>
<td>spring.activemq.pool.time-between-expiration-check</td>
<td>-1ms</td>
<td>Time to sleep between runs of the idle connection eviction thread. When negative, no idle connection eviction thread runs.</td>
</tr>
<tr>
<td>spring.activemq.pool.use-anonymous-producers</td>
<td>true</td>
<td>Whether to use only one anonymous &quot;MessageProducer&quot; instance. Set it to false to create one &quot;MessageProducer&quot; every time one is required.</td>
</tr>
<tr>
<td>spring.activemq.send-timeout</td>
<td>0ms</td>
<td>Time to wait on message sends for a response. Set it to 0 to wait forever.</td>
</tr>
<tr>
<td>spring.activemq.user</td>
<td></td>
<td>Login user of the broker.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.artemis.embedded.cluster-password</td>
<td></td>
<td>Cluster password. Randomly generated on startup by default.</td>
</tr>
<tr>
<td>spring.artemis.embedded.data-directory</td>
<td></td>
<td>Journal file directory. Not necessary if persistence is turned off.</td>
</tr>
<tr>
<td>spring.artemis.embedded.enabled</td>
<td>true</td>
<td>Whether to enable embedded mode if the Artemis server APIs are available.</td>
</tr>
<tr>
<td>spring.artemis.embedded.persistent</td>
<td>false</td>
<td>Whether to enable persistent store.</td>
</tr>
<tr>
<td>spring.artemis.embedded.queues</td>
<td>[]</td>
<td>Comma-separated list of queues to create on startup.</td>
</tr>
<tr>
<td>spring.artemis.embedded.server-id</td>
<td>0.0</td>
<td>Server ID. By default, an auto-incremented counter is used.</td>
</tr>
<tr>
<td>spring.artemis.embedded.topics</td>
<td>[]</td>
<td>Comma-separated list of topics to create on startup.</td>
</tr>
<tr>
<td>spring.artemis.host</td>
<td>localhost</td>
<td>Artemis broker host.</td>
</tr>
<tr>
<td>spring.artemis.mode</td>
<td></td>
<td>Artemis deployment mode, auto-detected by default.</td>
</tr>
<tr>
<td>spring.artemis.password</td>
<td></td>
<td>Login password of the broker.</td>
</tr>
<tr>
<td>spring.artemis.pool.block-if-full</td>
<td>true</td>
<td>Whether to block when a connection is requested and the pool is full. Set it to false to throw a &quot;JMSException&quot; instead.</td>
</tr>
<tr>
<td>spring.artemis.pool.block-if-full-timeout</td>
<td>-1ms</td>
<td>Blocking period before throwing an exception if the pool is still full.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.artemis.pool.enabled</td>
<td>false</td>
<td>Whether a JmsPoolConnectionFactory should be created, instead of a regular ConnectionFactory.</td>
</tr>
<tr>
<td>spring.artemis.pool.idle-timeout</td>
<td>30s</td>
<td>Connection idle timeout.</td>
</tr>
<tr>
<td>spring.artemis.pool.max-connections</td>
<td>1.0</td>
<td>Maximum number of pooled connections.</td>
</tr>
<tr>
<td>spring.artemis.pool.max-sessions-per-connection</td>
<td>500.0</td>
<td>Maximum number of pooled sessions per connection in the pool.</td>
</tr>
<tr>
<td>spring.artemis.pool.time-between-expiration-check</td>
<td>-1ms</td>
<td>Time to sleep between runs of the idle connection eviction thread. When negative, no idle connection eviction thread runs.</td>
</tr>
<tr>
<td>spring.artemis.pool.use-anonymous-producers</td>
<td>true</td>
<td>Whether to use only one anonymous MessageProducer instance. Set it to false to create one MessageProducer every time one is required.</td>
</tr>
<tr>
<td>spring.artemis.port</td>
<td>61616.0</td>
<td>Artemis broker port.</td>
</tr>
<tr>
<td>spring.artemis.user</td>
<td></td>
<td>Login user of the broker.</td>
</tr>
<tr>
<td>spring.batch.initialize-schema</td>
<td>embedded</td>
<td>Database schema initialization mode.</td>
</tr>
<tr>
<td>spring.batch.job.enabled</td>
<td>true</td>
<td>Execute all Spring Batch jobs in the context on startup.</td>
</tr>
<tr>
<td>spring.batch.job.names</td>
<td></td>
<td>Comma-separated list of job names to execute on startup (for instance, <code>job1,job2</code>). By default, all Jobs found in the context are executed.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.batch.schema</td>
<td>classpath:org/springframework/batch/core/schema-@@platform@@.sql</td>
<td>Path to the SQL file to use to initialize the database schema.</td>
</tr>
<tr>
<td>spring.batch.table-prefix</td>
<td></td>
<td>Table prefix for all the batch meta-data tables.</td>
</tr>
<tr>
<td>spring.hazelcast.config</td>
<td></td>
<td>The location of the configuration file to use to initialize Hazelcast.</td>
</tr>
<tr>
<td>spring.integration.jdbc.initialize-schema</td>
<td>embedded</td>
<td>Database schema initialization mode.</td>
</tr>
<tr>
<td>spring.integration.jdbc.schema</td>
<td>classpath:org/springframework/integration/jdbc/schema-@@platform@@.sql</td>
<td>Path to the SQL file to use to initialize the database schema.</td>
</tr>
<tr>
<td>spring.integration.rsocket.client.host</td>
<td></td>
<td>TCP RSocket server host to connect to.</td>
</tr>
<tr>
<td>spring.integration.rsocket.client.port</td>
<td></td>
<td>TCP RSocket server port to connect to.</td>
</tr>
<tr>
<td>spring.integration.rsocket.client.uri</td>
<td></td>
<td>WebSocket RSocket server uri to connect to.</td>
</tr>
<tr>
<td>spring.integration.rsocket.server.message-mapping-enabled</td>
<td>false</td>
<td>Whether to handle message mapping for RSocket via Spring Integration.</td>
</tr>
<tr>
<td>spring.jms.cache.consumers</td>
<td>false</td>
<td>Whether to cache message consumers.</td>
</tr>
<tr>
<td>spring.jms.cache.enabled</td>
<td>true</td>
<td>Whether to cache sessions.</td>
</tr>
<tr>
<td>spring.jms.cache.producers</td>
<td>true</td>
<td>Whether to cache message producers.</td>
</tr>
<tr>
<td>spring.jms.cache.session-cache-size</td>
<td>1.0</td>
<td>Size of the session cache (per JMS Session type).</td>
</tr>
<tr>
<td>spring.jms.jndi-name</td>
<td></td>
<td>Connection factory JNDI name. When set, takes precedence to others connection factory auto-configurations.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jms.listener.acknowledge-mode</td>
<td></td>
<td>Acknowledge mode of the container. By default, the listener is transacted with automatic acknowledgment.</td>
</tr>
<tr>
<td>spring.jms.listener.auto-startup</td>
<td>true</td>
<td>Start the container automatically on startup.</td>
</tr>
<tr>
<td>spring.jms.listener.concurrency</td>
<td></td>
<td>Minimum number of concurrent consumers.</td>
</tr>
<tr>
<td>spring.jms.listener.max-concurrency</td>
<td></td>
<td>Maximum number of concurrent consumers.</td>
</tr>
<tr>
<td>spring.jms.listener.receive-timeout</td>
<td>1s</td>
<td>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.</td>
</tr>
<tr>
<td>spring.jms.pub-sub-domain</td>
<td>false</td>
<td>Whether the default destination type is topic.</td>
</tr>
<tr>
<td>spring.jms.template.default-destination</td>
<td></td>
<td>Default destination to use on send and receive operations that do not have a destination parameter.</td>
</tr>
<tr>
<td>spring.jms.template.delivery-delay</td>
<td></td>
<td>Delivery delay to use for send calls.</td>
</tr>
<tr>
<td>spring.jms.template.delivery-mode</td>
<td></td>
<td>Delivery mode. Enables QoS (Quality of Service) when set.</td>
</tr>
<tr>
<td>spring.jms.template.priority</td>
<td></td>
<td>Priority of a message when sending. Enables QoS (Quality of Service) when set.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.jms.template.qos-enabled</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>spring.jms.template.receive-timeout</td>
<td></td>
<td>Timeout to use for receive calls.</td>
</tr>
<tr>
<td>spring.jms.template.time-to-live</td>
<td></td>
<td>Time-to-live of a message when sending. Enables QoS (Quality of Service) when set.</td>
</tr>
<tr>
<td>spring.kafka.admin.client-id</td>
<td></td>
<td>ID to pass to the server when making requests. Used for server-side logging.</td>
</tr>
<tr>
<td>spring.kafka.admin.fail-fast</td>
<td>false</td>
<td>Whether to fail fast if the broker is not available on startup.</td>
</tr>
<tr>
<td>spring.kafka.admin.properties.*</td>
<td></td>
<td>Additional admin-specific properties used to configure the client.</td>
</tr>
<tr>
<td>spring.kafka.admin.security.protocol</td>
<td></td>
<td>Security protocol used to communicate with brokers.</td>
</tr>
<tr>
<td>spring.kafka.admin.ssl.key-password</td>
<td></td>
<td>Password of the private key in the key store file.</td>
</tr>
<tr>
<td>spring.kafka.admin.ssl.key-store-location</td>
<td></td>
<td>Location of the key store file.</td>
</tr>
<tr>
<td>spring.kafka.admin.ssl.key-store-password</td>
<td></td>
<td>Store password for the key store file.</td>
</tr>
<tr>
<td>spring.kafka.admin.ssl.key-store-type</td>
<td></td>
<td>Type of the key store.</td>
</tr>
<tr>
<td>spring.kafka.admin.ssl.protocol</td>
<td></td>
<td>SSL protocol to use.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.kafka.admin.ssl.trust-store-location</td>
<td></td>
<td>Location of the trust store file.</td>
</tr>
<tr>
<td>spring.kafka.admin.ssl.trust-store-password</td>
<td></td>
<td>Store password for the trust store file.</td>
</tr>
<tr>
<td>spring.kafka.admin.ssl.trust-store-type</td>
<td></td>
<td>Type of the trust store.</td>
</tr>
<tr>
<td>spring.kafka.bootstrap-servers</td>
<td></td>
<td>Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Applies to all components unless overridden.</td>
</tr>
<tr>
<td>spring.kafka.client-id</td>
<td></td>
<td>ID to pass to the server when making requests. Used for server-side logging.</td>
</tr>
<tr>
<td>spring.kafka.consumer.auto-commit-interval</td>
<td></td>
<td>Frequency with which the consumer offsets are auto-committed to Kafka if 'enable.auto.commit' is set to true.</td>
</tr>
<tr>
<td>spring.kafka.consumer.auto-offset-reset</td>
<td></td>
<td>What to do when there is no initial offset in Kafka or if the current offset no longer exists on the server.</td>
</tr>
<tr>
<td>spring.kafka.consumer.bootstrap-servers</td>
<td></td>
<td>Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Overrides the global property, for consumers.</td>
</tr>
<tr>
<td>spring.kafka.consumer.client-id</td>
<td></td>
<td>ID to pass to the server when making requests. Used for server-side logging.</td>
</tr>
</tbody>
</table>

479
<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.kafka.consumer.enable-auto-commit</td>
<td></td>
<td>Whether the consumer's offset is periodically committed in the background.</td>
</tr>
<tr>
<td>spring.kafka.consumer.fetch-max-wait</td>
<td></td>
<td>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”.</td>
</tr>
<tr>
<td>spring.kafka.consumer.fetch-min-size</td>
<td></td>
<td>Minimum amount of data the server should return for a fetch request.</td>
</tr>
<tr>
<td>spring.kafka.consumer.group-id</td>
<td></td>
<td>Unique string that identifies the consumer group to which this consumer belongs.</td>
</tr>
<tr>
<td>spring.kafka.consumer.heartbeat-interval</td>
<td></td>
<td>Expected time between heartbeats to the consumer coordinator.</td>
</tr>
<tr>
<td>spring.kafka.consumer.isolation-level</td>
<td>read-uncommitted</td>
<td>Isolation level for reading messages that have been written transactionally.</td>
</tr>
<tr>
<td>spring.kafka.consumer.key-deserializer</td>
<td></td>
<td>Deserializer class for keys.</td>
</tr>
<tr>
<td>spring.kafka.consumer.max-poll-records</td>
<td></td>
<td>Maximum number of records returned in a single call to poll().</td>
</tr>
<tr>
<td>spring.kafka.consumer.properties.*</td>
<td></td>
<td>Additional consumer-specific properties used to configure the client.</td>
</tr>
<tr>
<td>spring.kafka.consumer.security.protocol</td>
<td></td>
<td>Security protocol used to communicate with brokers.</td>
</tr>
<tr>
<td>spring.kafka.consumer.ssl.key-password</td>
<td></td>
<td>Password of the private key in the key store file.</td>
</tr>
<tr>
<td>spring.kafka.consumer.ssl.key-store-location</td>
<td></td>
<td>Location of the key store file.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>spring.kafka.consumer.ssl.key-store-password</td>
<td></td>
<td>Store password for the key store file.</td>
</tr>
<tr>
<td>spring.kafka.consumer.ssl.key-store-type</td>
<td></td>
<td>Type of the key store.</td>
</tr>
<tr>
<td>spring.kafka.consumer.ssl.protocol</td>
<td></td>
<td>SSL protocol to use.</td>
</tr>
<tr>
<td>spring.kafka.consumer.ssl.trust-store-location</td>
<td></td>
<td>Location of the trust store file.</td>
</tr>
<tr>
<td>spring.kafka.consumer.ssl.trust-store-password</td>
<td></td>
<td>Store password for the trust store file.</td>
</tr>
<tr>
<td>spring.kafka.consumer.ssl.trust-store-type</td>
<td></td>
<td>Type of the trust store.</td>
</tr>
<tr>
<td>spring.kafka.consumer.value-deserializer</td>
<td></td>
<td>Deserializer class for values.</td>
</tr>
<tr>
<td>spring.kafka.jaas.control-flag</td>
<td>required</td>
<td>Control flag for login configuration.</td>
</tr>
<tr>
<td>spring.kafka.jaas.enabled</td>
<td>false</td>
<td>Whether to enable JAAS configuration.</td>
</tr>
<tr>
<td>spring.kafka.jaas.options.*</td>
<td></td>
<td>Additional JAAS options.</td>
</tr>
<tr>
<td>spring.kafka.listener.ack-count</td>
<td></td>
<td>Number of records between offset commits when ackMode is &quot;COUNT&quot; or &quot;COUNT_TIME&quot;.</td>
</tr>
<tr>
<td>spring.kafka.listener.ack-mode</td>
<td></td>
<td>Listener AckMode. See the spring-kafka documentation.</td>
</tr>
<tr>
<td>spring.kafka.listener.ack-time</td>
<td></td>
<td>Time between offset commits when ackMode is &quot;TIME&quot; or &quot;COUNT_TIME&quot;.</td>
</tr>
<tr>
<td>spring.kafka.listener.client-id</td>
<td></td>
<td>Prefix for the listener's consumer client.id property.</td>
</tr>
<tr>
<td>spring.kafka.listener.concurrency</td>
<td></td>
<td>Number of threads to run in the listener containers.</td>
</tr>
<tr>
<td>spring.kafka.listener.idle-between-polls</td>
<td>0.0</td>
<td>Sleep interval between Consumer.poll(Duration) calls.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.kafka.listener.idle-event-interval</td>
<td></td>
<td>Time between publishing idle consumer events (no data received).</td>
</tr>
<tr>
<td>spring.kafka.listener.log-container-config</td>
<td></td>
<td>Whether to log the container configuration during initialization (INFO level).</td>
</tr>
<tr>
<td>spring.kafka.listener.missing-topics-fatal</td>
<td>false</td>
<td>Whether the container should fail to start if at least one of the configured topics are not present on the broker.</td>
</tr>
<tr>
<td>spring.kafka.listener.monitor-interval</td>
<td></td>
<td>Time between checks for non-responsive consumers. If a duration suffix is not specified, seconds will be used.</td>
</tr>
<tr>
<td>spring.kafka.listener.no-poll-threshold</td>
<td></td>
<td>Multiplier applied to &quot;pollTimeout&quot; to determine if a consumer is non-responsive.</td>
</tr>
<tr>
<td>spring.kafka.listener.poll-timeout</td>
<td></td>
<td>Timeout to use when polling the consumer.</td>
</tr>
<tr>
<td>spring.kafka.listener.type</td>
<td>single</td>
<td>Listener type.</td>
</tr>
<tr>
<td>spring.kafka.producer.acks</td>
<td></td>
<td>Number of acknowledgments the producer requires the leader to have received before considering a request complete.</td>
</tr>
<tr>
<td>spring.kafka.producer.batch-size</td>
<td></td>
<td>Default batch size. A small batch size will make batching less common and may reduce throughput (a batch size of zero disables batching entirely).</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>-------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.kafka.producer.bootstrap-servers</td>
<td></td>
<td>Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Overrides the global property, for producers.</td>
</tr>
<tr>
<td>spring.kafka.producer.buffer-memory</td>
<td></td>
<td>Total memory size the producer can use to buffer records waiting to be sent to the server.</td>
</tr>
<tr>
<td>spring.kafka.producer.client-id</td>
<td></td>
<td>ID to pass to the server when making requests. Used for server-side logging.</td>
</tr>
<tr>
<td>spring.kafka.producer.compression-type</td>
<td></td>
<td>Compression type for all data generated by the producer.</td>
</tr>
<tr>
<td>spring.kafka.producer.key-serializer</td>
<td></td>
<td>Serializer class for keys.</td>
</tr>
<tr>
<td>spring.kafka.producer.properties.*</td>
<td></td>
<td>Additional producer-specific properties used to configure the client.</td>
</tr>
<tr>
<td>spring.kafka.producer.retries</td>
<td></td>
<td>When greater than zero, enables retrying of failed sends.</td>
</tr>
<tr>
<td>spring.kafka.producer.security.protocol</td>
<td></td>
<td>Security protocol used to communicate with brokers.</td>
</tr>
<tr>
<td>spring.kafka.producer.ssl.key-password</td>
<td></td>
<td>Password of the private key in the key store file.</td>
</tr>
<tr>
<td>spring.kafka.producer.ssl.key-store-location</td>
<td></td>
<td>Location of the key store file.</td>
</tr>
<tr>
<td>spring.kafka.producer.ssl.key-store-password</td>
<td></td>
<td>Store password for the key store file.</td>
</tr>
<tr>
<td>spring.kafka.producer.ssl.key-store-type</td>
<td></td>
<td>Type of the key store.</td>
</tr>
<tr>
<td>spring.kafka.producer.ssl.protocol</td>
<td></td>
<td>SSL protocol to use.</td>
</tr>
<tr>
<td>spring.kafka.producer.ssl.trust-store-location</td>
<td></td>
<td>Location of the trust store file.</td>
</tr>
<tr>
<td>spring.kafka.producer.ssl.trust-store-password</td>
<td></td>
<td>Store password for the trust store file.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>spring.kafka.producer.ssl.trust-store-type</td>
<td></td>
<td>Type of the trust store.</td>
</tr>
<tr>
<td>spring.kafka.producer.transaction-id-prefix</td>
<td></td>
<td>When non empty, enables transaction support for producer.</td>
</tr>
<tr>
<td>spring.kafka.producer.value-serializer</td>
<td></td>
<td>Serializer class for values.</td>
</tr>
<tr>
<td>spring.kafka.properties.*</td>
<td></td>
<td>Additional properties, common to producers and consumers, used to configure the client.</td>
</tr>
<tr>
<td>spring.kafka.security.protocol</td>
<td></td>
<td>Security protocol used to communicate with brokers.</td>
</tr>
<tr>
<td>spring.kafka.ssl.key-password</td>
<td></td>
<td>Password of the private key in the key store file.</td>
</tr>
<tr>
<td>spring.kafka.ssl.key-store-location</td>
<td></td>
<td>Location of the key store file.</td>
</tr>
<tr>
<td>spring.kafka.ssl.key-store-password</td>
<td></td>
<td>Store password for the key store file.</td>
</tr>
<tr>
<td>spring.kafka.ssl.key-store-type</td>
<td></td>
<td>Type of the key store.</td>
</tr>
<tr>
<td>spring.kafka.ssl.protocol</td>
<td></td>
<td>SSL protocol to use.</td>
</tr>
<tr>
<td>spring.kafka.ssl.trust-store-location</td>
<td></td>
<td>Location of the trust store file.</td>
</tr>
<tr>
<td>spring.kafka.ssl.trust-store-password</td>
<td></td>
<td>Store password for the trust store file.</td>
</tr>
<tr>
<td>spring.kafka.ssl.trust-store-type</td>
<td></td>
<td>Type of the trust store.</td>
</tr>
<tr>
<td>spring.kafka.streams.application-id</td>
<td></td>
<td>Kafka streams application.id property; default spring.application.name.</td>
</tr>
<tr>
<td>spring.kafka.streams.auto-startup</td>
<td>true</td>
<td>Whether or not to auto-start the streams factory bean.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.kafka.streams.bootstrap-servers</code></td>
<td></td>
<td>Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster. Overrides the global property, for streams.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.cache-max-size-buffering</code></td>
<td></td>
<td>Maximum memory size to be used for buffering across all threads.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.cleanup.on-shutdown</code></td>
<td>true</td>
<td>Cleanup the application's local state directory on shutdown.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.cleanup.on-startup</code></td>
<td>false</td>
<td>Cleanup the application’s local state directory on startup.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.client-id</code></td>
<td></td>
<td>ID to pass to the server when making requests. Used for server-side logging.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.properties.*</code></td>
<td></td>
<td>Additional Kafka properties used to configure the streams.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.replication-factor</code></td>
<td></td>
<td>The replication factor for change log topics and repartition topics created by the stream processing application.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.security.protocol</code></td>
<td></td>
<td>Security protocol used to communicate with brokers.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.ssl.key-password</code></td>
<td></td>
<td>Password of the private key in the key store file.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.ssl.key-store-location</code></td>
<td></td>
<td>Location of the key store file.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.ssl.key-store-password</code></td>
<td></td>
<td>Store password for the key store file.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.ssl.key-store-type</code></td>
<td></td>
<td>Type of the key store.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.ssl.protocol</code></td>
<td></td>
<td>SSL protocol to use.</td>
</tr>
<tr>
<td><code>spring.kafka.streams.ssl.trust-store-location</code></td>
<td></td>
<td>Location of the trust store file.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.kafka.streams.ssl.trust-store-password</td>
<td></td>
<td>Store password for the trust store file.</td>
</tr>
<tr>
<td>spring.kafka.streams.ssl.trust-store-type</td>
<td></td>
<td>Type of the trust store.</td>
</tr>
<tr>
<td>spring.kafka.streams.state-dir</td>
<td></td>
<td>Directory location for the state store.</td>
</tr>
<tr>
<td>spring.kafka.template.default-topic</td>
<td></td>
<td>Default topic to which messages are sent.</td>
</tr>
<tr>
<td>spring.rabbitmq.address-shuffle-mode</td>
<td>none</td>
<td>Mode used to shuffle configured addresses.</td>
</tr>
<tr>
<td>spring.rabbitmq.addresses</td>
<td></td>
<td>Comma-separated list of addresses to which the client should connect.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, the host and port are ignored.</td>
</tr>
<tr>
<td>spring.rabbitmq.cache.channel.checkout-timeout</td>
<td></td>
<td>Duration to wait to obtain a channel if the cache size has been reached.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If 0, always create a new channel.</td>
</tr>
<tr>
<td>spring.rabbitmq.cache.channel.size</td>
<td></td>
<td>Number of channels to retain in the cache.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When &quot;check-timeout&quot; &gt; 0, max channels per connection.</td>
</tr>
<tr>
<td>spring.rabbitmq.cache.connection.mode</td>
<td>channel</td>
<td>Connection factory cache mode.</td>
</tr>
<tr>
<td>spring.rabbitmq.cache.connection.size</td>
<td></td>
<td>Number of connections to cache. Only applies when mode is CONNECTION.</td>
</tr>
<tr>
<td>spring.rabbitmq.channel-rpc-timeout</td>
<td>10m</td>
<td>Continuation timeout for RPC calls in channels. Set it to zero to wait</td>
</tr>
<tr>
<td></td>
<td></td>
<td>forever.</td>
</tr>
<tr>
<td>spring.rabbitmq.connection-timeout</td>
<td></td>
<td>Connection timeout. Set it to zero to wait forever.</td>
</tr>
<tr>
<td>spring.rabbitmq.dynamic</td>
<td>true</td>
<td>Whether to create an AmqpAdmin bean.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.rabbitmq.host</td>
<td>localhost</td>
<td>RabbitMQ host. Ignored if an address is set.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.acknowledge-mode</td>
<td></td>
<td>Acknowledge mode of container.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.auto-startup</td>
<td>true</td>
<td>Whether to start the container automatically on startup.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.consumers-per-queue</td>
<td></td>
<td>Number of consumers per queue.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.de-batching-enabled</td>
<td>true</td>
<td>Whether the container should present batched messages as discrete messages or call the listener with the batch.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.default-requeue-rejected</td>
<td></td>
<td>Whether rejected deliveries are re-queued by default.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.idle-event-interval</td>
<td></td>
<td>How often idle container events should be published.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.missing-queues-fatal</td>
<td>false</td>
<td>Whether to fail if the queues declared by the container are not available on the broker.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.prefetch</td>
<td></td>
<td>Maximum number of unacknowledged messages that can be outstanding at each consumer.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.retry.enabled</td>
<td>false</td>
<td>Whether publishing retries are enabled.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.retry.initial-interval</td>
<td>1000ms</td>
<td>Duration between the first and second attempt to deliver a message.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.retry.max-attempts</td>
<td>3.0</td>
<td>Maximum number of attempts to deliver a message.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.retry.max-interval</td>
<td>10000ms</td>
<td>Maximum duration between attempts.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.retry.multiplier</td>
<td>1.0</td>
<td>Multiplier to apply to the previous retry interval.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.direct.retry.stateless</td>
<td>true</td>
<td>Whether retries are stateless or stateful.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.acknowledge-mode</td>
<td></td>
<td>Acknowledge mode of container.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.auto-startup</td>
<td>true</td>
<td>Whether to start the container automatically on startup.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.batch-size</td>
<td></td>
<td>Batch size, expressed as the number of physical messages, to be used by the container.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.concurrency</td>
<td></td>
<td>Minimum number of listener invoker threads.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.consumer-batch-enabled</td>
<td>false</td>
<td>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.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.de-batching-enabled</td>
<td>true</td>
<td>Whether the container should present batched messages as discrete messages or call the listener with the batch.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.default-requeue-rejected</td>
<td></td>
<td>Whether rejected deliveries are re-queued by default.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.idle-event-interval</td>
<td></td>
<td>How often idle container events should be published.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.max-concurrency</td>
<td></td>
<td>Maximum number of listener invoker threads.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.missing-queues-fatal</td>
<td>true</td>
<td>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.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.prefetch</td>
<td></td>
<td>Maximum number of unacknowledged messages that can be outstanding at each consumer.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.retry.enabled</td>
<td>false</td>
<td>Whether publishing retries are enabled.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.retry.initial-interval</td>
<td>1000ms</td>
<td>Duration between the first and second attempt to deliver a message.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.retry.max-attempts</td>
<td>3.0</td>
<td>Maximum number of attempts to deliver a message.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.retry.max-interval</td>
<td>10000ms</td>
<td>Maximum duration between attempts.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.retry.multiplier</td>
<td>1.0</td>
<td>Multiplier to apply to the previous retry interval.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.simple.retry.stateless</td>
<td>true</td>
<td>Whether retries are stateless or stateful.</td>
</tr>
<tr>
<td>spring.rabbitmq.listener.type</td>
<td>simple</td>
<td>Listener container type.</td>
</tr>
<tr>
<td>spring.rabbitmq.password</td>
<td>guest</td>
<td>Login to authenticate against the broker.</td>
</tr>
<tr>
<td>spring.rabbitmq.port</td>
<td></td>
<td>RabbitMQ port. Ignored if an address is set. Default to 5672, or 5671 if SSL is enabled.</td>
</tr>
<tr>
<td>spring.rabbitmq.publisher-confirm-type</td>
<td></td>
<td>Type of publisher confirms to use.</td>
</tr>
<tr>
<td>spring.rabbitmq.publisher-returns</td>
<td>false</td>
<td>Whether to enable publisher returns.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.rabbitmq.requested-channel-max</code></td>
<td>2047.0</td>
<td>Number of channels per connection requested by the client. Use 0 for unlimited.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.requested-heartbeat</code></td>
<td></td>
<td>Requested heartbeat timeout; zero for none. If a duration suffix is not specified, seconds will be used.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.algorithm</code></td>
<td></td>
<td>SSL algorithm to use. By default, configured by the Rabbit client library.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.enabled</code></td>
<td></td>
<td>Whether to enable SSL support. Determined automatically if an address is provided with the protocol (amqp:// vs. amqps://).</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.key-store</code></td>
<td></td>
<td>Path to the key store that holds the SSL certificate.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.key-store-password</code></td>
<td></td>
<td>Password used to access the key store.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.key-store-type</code></td>
<td>PKCS12</td>
<td>Key store type.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.trust-store</code></td>
<td></td>
<td>Trust store that holds SSL certificates.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.trust-store-password</code></td>
<td></td>
<td>Password used to access the trust store.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.trust-store-type</code></td>
<td>JKS</td>
<td>Trust store type.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.validate-server-certificate</code></td>
<td>true</td>
<td>Whether to enable server side certificate validation.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.ssl.verify-hostname</code></td>
<td>true</td>
<td>Whether to enable hostname verification.</td>
</tr>
<tr>
<td><code>spring.rabbitmq.template.default-receive-queue</code></td>
<td></td>
<td>Name of the default queue to receive messages from when none is specified explicitly.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.rabbitmq.template.exchange</td>
<td></td>
<td>Name of the default exchange to use for send operations.</td>
</tr>
<tr>
<td>spring.rabbitmq.template.mandatory</td>
<td></td>
<td>Whether to enable mandatory messages.</td>
</tr>
<tr>
<td>spring.rabbitmq.template.receive-timeout</td>
<td></td>
<td>Timeout for <code>receive()</code> operations.</td>
</tr>
<tr>
<td>spring.rabbitmq.template.reply-timeout</td>
<td></td>
<td>Timeout for <code>sendAndReceive()</code> operations.</td>
</tr>
<tr>
<td>spring.rabbitmq.template.retry.enabled</td>
<td>false</td>
<td>Whether publishing retries are enabled.</td>
</tr>
<tr>
<td>spring.rabbitmq.template.retry.initial-interval</td>
<td>1000ms</td>
<td>Duration between the first and second attempt to deliver a message.</td>
</tr>
<tr>
<td>spring.rabbitmq.template.retry.max-attempts</td>
<td>3.0</td>
<td>Maximum number of attempts to deliver a message.</td>
</tr>
<tr>
<td>spring.rabbitmq.template.retry.max-interval</td>
<td>10000ms</td>
<td>Maximum duration between attempts.</td>
</tr>
<tr>
<td>spring.rabbitmq.template.retry.multiplier</td>
<td>1.0</td>
<td>Multiplier to apply to the previous retry interval.</td>
</tr>
<tr>
<td>spring.rabbitmq.template.routing-key</td>
<td></td>
<td>Value of a default routing key to use for send operations.</td>
</tr>
<tr>
<td>spring.rabbitmq.username</td>
<td>guest</td>
<td>Login user to authenticate to the broker.</td>
</tr>
<tr>
<td>spring.rabbitmq.virtual-host</td>
<td></td>
<td>Virtual host to use when connecting to the broker.</td>
</tr>
</tbody>
</table>

**10.A.9. Web Properties**
<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.hateoas.use-hal-as-default-json-media-type</code></td>
<td><code>true</code></td>
<td>Whether application/hal+json responses should be sent to requests that accept application/json.</td>
</tr>
<tr>
<td><code>spring.jersey.application-path</code></td>
<td></td>
<td>Path that serves as the base URI for the application. If specified, overrides the value of &quot;@ApplicationPath&quot;.</td>
</tr>
<tr>
<td><code>spring.jersey.filter.order</code></td>
<td><code>0.0</code></td>
<td>Jersey filter chain order.</td>
</tr>
<tr>
<td><code>spring.jersey.init.*</code></td>
<td></td>
<td>Init parameters to pass to Jersey through the servlet or filter.</td>
</tr>
<tr>
<td><code>spring.jersey.servlet.load-on-startup</code></td>
<td><code>-1.0</code></td>
<td>Load on startup priority of the Jersey servlet.</td>
</tr>
<tr>
<td><code>spring.jersey.type</code></td>
<td><code>servlet</code></td>
<td>Jersey integration type.</td>
</tr>
<tr>
<td><code>spring.mvc.async.request-timeout</code></td>
<td></td>
<td>Amount of time before asynchronous request handling times out. If this value is not set, the default timeout of the underlying implementation is used.</td>
</tr>
<tr>
<td><code>spring.mvc.contentnegotiation.favor-parameter</code></td>
<td><code>false</code></td>
<td>Whether a request parameter (&quot;format&quot; by default) should be used to determine the requested media type.</td>
</tr>
<tr>
<td><code>spring.mvc.contentnegotiation.media-types.*</code></td>
<td></td>
<td>Map file extensions to media types for content negotiation. For instance, yml to text/yaml.</td>
</tr>
<tr>
<td><code>spring.mvc.contentnegotiation.parameter-name</code></td>
<td></td>
<td>Query parameter name to use when &quot;favor-parameter&quot; is enabled.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.mvc.converters.preferred-json-mapper</td>
<td></td>
<td>Preferred JSON mapper to use for HTTP message conversion. By default, auto-detected according to the environment.</td>
</tr>
<tr>
<td>spring.mvc.dispatch-options-request</td>
<td>true</td>
<td>Whether to dispatch OPTIONS requests to the FrameworkServlet doService method.</td>
</tr>
<tr>
<td>spring.mvc.dispatch-trace-request</td>
<td>false</td>
<td>Whether to dispatch TRACE requests to the FrameworkServlet doService method.</td>
</tr>
<tr>
<td>spring.mvc.format.date</td>
<td></td>
<td>Date format to use, for example <code>dd/MM/yyyy</code>.</td>
</tr>
<tr>
<td>spring.mvc.format.date-time</td>
<td></td>
<td>Date-time format to use, for example <code>yyyy-MM-dd HH:mm:ss</code>.</td>
</tr>
<tr>
<td>spring.mvc.format.time</td>
<td></td>
<td>Time format to use, for example <code>HH:mm:ss</code>.</td>
</tr>
<tr>
<td>spring.mvc.formcontent.filter.enabled</td>
<td>true</td>
<td>Whether to enable Spring's FormContentFilter.</td>
</tr>
<tr>
<td>spring.mvc.hiddenmethod.filter.enabled</td>
<td>false</td>
<td>Whether to enable Spring's HiddenHttpMethodFilter.</td>
</tr>
<tr>
<td>spring.mvc.ignore-default-model-on-redirect</td>
<td>true</td>
<td>Whether the content of the &quot;default&quot; model should be ignored during redirect scenarios.</td>
</tr>
<tr>
<td>spring.mvc.log-request-details</td>
<td>false</td>
<td>Whether logging of (potentially sensitive) request details at DEBUG and TRACE level is allowed.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.mvc.log-resolved-exception</code></td>
<td><code>false</code></td>
<td>Whether to enable warn logging of exceptions resolved by a &quot;HandlerExceptionResolver&quot;, except for &quot;DefaultHandlerExceptionResolver&quot;.</td>
</tr>
<tr>
<td><code>spring.mvc.message-codes-resolver-format</code></td>
<td></td>
<td>Formatting strategy for message codes. For instance, <code>PREFIX_ERROR_CODE</code>.</td>
</tr>
<tr>
<td><code>spring.mvc.pathmatch.matching-strategy</code></td>
<td><code>ant-path-matcher</code></td>
<td>Choice of strategy for matching request paths against registered mappings.</td>
</tr>
<tr>
<td><code>spring.mvc.publish-request-handled-events</code></td>
<td><code>true</code></td>
<td>Whether to publish a ServletRequestHandledEvent at the end of each request.</td>
</tr>
<tr>
<td><code>spring.mvc.servlet.load-on-startup</code></td>
<td><code>-1.0</code></td>
<td>Load on startup priority of the dispatcher servlet.</td>
</tr>
<tr>
<td><code>spring.mvc.servlet.path</code></td>
<td><code>/</code></td>
<td>Path of the dispatcher servlet. Setting a custom value for this property is not compatible with the PathPatternParser matching strategy.</td>
</tr>
<tr>
<td><code>spring.mvc.static-path-pattern</code></td>
<td><code>/**</code></td>
<td>Path pattern used for static resources.</td>
</tr>
<tr>
<td><code>spring.mvc.throw-exception-if-no-handler-found</code></td>
<td><code>false</code></td>
<td>Whether a &quot;NoHandlerFoundException&quot; should be thrown if no Handler was found to process a request.</td>
</tr>
<tr>
<td><code>spring.mvc.view.prefix</code></td>
<td></td>
<td>Spring MVC view prefix.</td>
</tr>
<tr>
<td><code>spring.mvc.view.suffix</code></td>
<td></td>
<td>Spring MVC view suffix.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.servlet.multipart.enabled</td>
<td>true</td>
<td>Whether to enable support of multipart uploads.</td>
</tr>
<tr>
<td>spring.servlet.multipart.file-size-threshold</td>
<td>0B</td>
<td>Threshold after which files are written to disk.</td>
</tr>
<tr>
<td>spring.servlet.multipart.location</td>
<td></td>
<td>Intermediate location of uploaded files.</td>
</tr>
<tr>
<td>spring.servlet.multipart.max-file-size</td>
<td>1MB</td>
<td>Max file size.</td>
</tr>
<tr>
<td>spring.servlet.multipart.max-request-size</td>
<td>10MB</td>
<td>Max request size.</td>
</tr>
<tr>
<td>spring.servlet.multipart.resolve-lazily</td>
<td>false</td>
<td>Whether to resolve the multipart request lazily at the time of file or parameter access.</td>
</tr>
<tr>
<td>spring.session.hazelcast.flush-mode</td>
<td>on-save</td>
<td>Sessions flush mode. Determines when session changes are written to the session store.</td>
</tr>
<tr>
<td>spring.session.hazelcast.map-name</td>
<td>spring:session:session</td>
<td>Name of the map used to store sessions.</td>
</tr>
<tr>
<td>spring.session.hazelcast.save-mode</td>
<td>on-set-attribute</td>
<td>Sessions save mode. Determines how session changes are tracked and saved to the session store.</td>
</tr>
<tr>
<td>spring.session.jdbc.cleanup-cron</td>
<td>0 * * * * *</td>
<td>Cron expression for expired session cleanup job.</td>
</tr>
<tr>
<td>spring.session.jdbc.flush-mode</td>
<td>on-save</td>
<td>Sessions flush mode. Determines when session changes are written to the session store.</td>
</tr>
<tr>
<td>spring.session.jdbc.initialize-schema</td>
<td>embedded</td>
<td>Database schema initialization mode.</td>
</tr>
<tr>
<td>spring.session.jdbc.save-mode</td>
<td>on-set-attribute</td>
<td>Sessions save mode. Determines how session changes are tracked and saved to the session store.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>----------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.session.jdbc.schema</td>
<td>classpath:org/springframework/session/jdbc/schema-@@platform@@.sql</td>
<td>Path to the SQL file to use to initialize the database schema.</td>
</tr>
<tr>
<td>spring.session.jdbc.table-name</td>
<td>SPRING_SESSION</td>
<td>Name of the database table used to store sessions.</td>
</tr>
<tr>
<td>spring.session.mongodb.collection-name</td>
<td>sessions</td>
<td>Collection name used to store sessions.</td>
</tr>
<tr>
<td>spring.session.redis.cleanup-cron</td>
<td>0 * * * * *</td>
<td>Cron expression for expired session cleanup job.</td>
</tr>
<tr>
<td>spring.session.redis.configure-action</td>
<td>notify-keyspace-events</td>
<td>The configure action to apply when no user defined ConfigureRedisAction bean is present.</td>
</tr>
<tr>
<td>spring.session.redis.flush-mode</td>
<td>on-save</td>
<td>Sessions flush mode. Determines when session changes are written to the session store.</td>
</tr>
<tr>
<td>spring.session.redis.namespace</td>
<td>spring:session</td>
<td>Namespace for keys used to store sessions.</td>
</tr>
<tr>
<td>spring.session.redis.save-mode</td>
<td>on-set-attribute</td>
<td>Sessions save mode. Determines how session changes are tracked and saved to the session store.</td>
</tr>
<tr>
<td>spring.session.servlet.filter-dispatcher-types</td>
<td>[async, error, request]</td>
<td>Session repository filter dispatcher types.</td>
</tr>
<tr>
<td>spring.session.servlet.filter-order</td>
<td></td>
<td>Session repository filter order.</td>
</tr>
<tr>
<td>spring.session.store-type</td>
<td></td>
<td>Session store type.</td>
</tr>
<tr>
<td>spring.session.timeout</td>
<td></td>
<td>Session timeout. If a duration suffix is not specified, seconds will be used.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.web.locale</td>
<td></td>
<td>Locale to use. By default, this locale is overridden by the &quot;Accept-Language&quot; header.</td>
</tr>
<tr>
<td>spring.web.locale-resolver</td>
<td>accept-header</td>
<td>Define how the locale should be resolved.</td>
</tr>
<tr>
<td>spring.web.resources.add-mappings</td>
<td>true</td>
<td>Whether to enable default resource handling.</td>
</tr>
<tr>
<td>spring.web.resources.cache.cachecontrol.cache-private</td>
<td></td>
<td>Indicate that the response message is intended for a single user and must not be stored by a shared cache.</td>
</tr>
<tr>
<td>spring.web.resources.cache.cachecontrol.cache-public</td>
<td></td>
<td>Indicate that any cache may store the response.</td>
</tr>
<tr>
<td>spring.web.resources.cache.cachecontrol.max-age</td>
<td></td>
<td>Maximum time the response should be cached, in seconds if no duration suffix is not specified.</td>
</tr>
<tr>
<td>spring.web.resources.cache.cachecontrol.must-revalidate</td>
<td></td>
<td>Indicate that once it has become stale, a cache must not use the response without revalidating it with the server.</td>
</tr>
<tr>
<td>spring.web.resources.cache.cachecontrol.no-cache</td>
<td></td>
<td>Indicate that the cached response can be reused only if revalidated with the server.</td>
</tr>
<tr>
<td>spring.web.resources.cache.cachecontrol.no-store</td>
<td></td>
<td>Indicate to not cache the response in any case.</td>
</tr>
<tr>
<td>spring.web.resources.cache.cachecontrol.no-transform</td>
<td></td>
<td>Indicate intermediaries (caches and others) that they should not transform the response content.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>spring.web.resources.cache.cachecontrol.proxy-revalidate</code></td>
<td></td>
<td>Same meaning as the &quot;must-revalidate&quot; directive, except that it does not apply to private caches.</td>
</tr>
<tr>
<td><code>spring.web.resources.cache.cachecontrol.s-max-age</code></td>
<td></td>
<td>Maximum time the response should be cached by shared caches, in seconds if no duration suffix is not specified.</td>
</tr>
<tr>
<td><code>spring.web.resources.cache.cachecontrol.stale-if-error</code></td>
<td></td>
<td>Maximum time the response may be used when errors are encountered, in seconds if no duration suffix is not specified.</td>
</tr>
<tr>
<td><code>spring.web.resources.cache.cachecontrol.stale-while-revalidate</code></td>
<td></td>
<td>Maximum time the response can be served after it becomes stale, in seconds if no duration suffix is not specified.</td>
</tr>
<tr>
<td><code>spring.web.resources.cache.period</code></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><code>spring.web.resources.cache.use-last-modified</code></td>
<td>true</td>
<td>Whether we should use the &quot;lastModified&quot; metadata of the files in HTTP caching headers.</td>
</tr>
<tr>
<td><code>spring.web.resources.chain.cache</code></td>
<td>true</td>
<td>Whether to enable caching in the Resource chain.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>spring.web.resources.chain.compressed</td>
<td>false</td>
<td>Whether to enable resolution of already compressed resources (gzip, brotli). Checks for a resource name with the '.gz' or '.br' file extensions.</td>
</tr>
<tr>
<td>spring.web.resources.chain.enabled</td>
<td></td>
<td>Whether to enable the Spring Resource Handling chain. By default, disabled unless at least one strategy has been enabled.</td>
</tr>
<tr>
<td>spring.web.resources.chain.strategy.content.enabled</td>
<td>false</td>
<td>Whether to enable the content Version Strategy.</td>
</tr>
<tr>
<td>spring.web.resources.chain.strategy.content.paths</td>
<td>[/**]</td>
<td>Comma-separated list of patterns to apply to the content Version Strategy.</td>
</tr>
<tr>
<td>spring.web.resources.chain.strategy.fixed.enabled</td>
<td>false</td>
<td>Whether to enable the fixed Version Strategy.</td>
</tr>
<tr>
<td>spring.web.resources.chain.strategy.fixed.paths</td>
<td>[/**]</td>
<td>Comma-separated list of patterns to apply to the fixed Version Strategy.</td>
</tr>
<tr>
<td>spring.web.resources.chain.strategy.fixed.version</td>
<td></td>
<td>Version string to use for the fixed Version Strategy.</td>
</tr>
<tr>
<td>spring.web.resources.static-locations</td>
<td>[classpath:/META-INF/resources/, classpath:/resources/, classpath:/static/, classpath:/public/]</td>
<td>Locations of static resources. Defaults to classpath:/META-INF/resources/, /resources/, /static/, /public/.</td>
</tr>
<tr>
<td>spring.webflux.base-path</td>
<td></td>
<td>Base path for all web handlers.</td>
</tr>
<tr>
<td>spring.webflux.format.date</td>
<td></td>
<td>Date format to use, for example <code>dd/MM/yyyy</code>.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.webflux.format.date-time</td>
<td></td>
<td>Date-time format to use, for example <code>yyyy-MM-dd HH:mm:ss</code>.</td>
</tr>
<tr>
<td>spring.webflux.format.time</td>
<td></td>
<td>Time format to use, for example <code>HH:mm:ss</code>.</td>
</tr>
<tr>
<td>spring.webflux.hiddenmethod.filter.enabled</td>
<td>false</td>
<td>Whether to enable Spring's HiddenHttpMethodFilter.</td>
</tr>
<tr>
<td>spring.webflux.static-path-pattern</td>
<td>/**</td>
<td>Path pattern used for static resources.</td>
</tr>
<tr>
<td>spring.webservices.path</td>
<td>/services</td>
<td>Path that serves as the base URI for the services.</td>
</tr>
<tr>
<td>spring.webservices.servlet.init.*</td>
<td></td>
<td>Servlet init parameters to pass to Spring Web Services.</td>
</tr>
<tr>
<td>spring.webservices.servlet.load-on-startup</td>
<td>-1.0</td>
<td>Load on startup priority of the Spring Web Services servlet.</td>
</tr>
<tr>
<td>spring.webservices.wsdl-locations</td>
<td></td>
<td>Comma-separated list of locations of WSDLs and accompanying XSDs to be exposed as beans.</td>
</tr>
</tbody>
</table>

**10.A.10. Templating Properties**

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.freemarker.allow-request-override</td>
<td>false</td>
<td>Whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.</td>
</tr>
<tr>
<td>spring.freemarker.allow-session-override</td>
<td>false</td>
<td>Whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.freemarker.cache</td>
<td>false</td>
<td>Whether to enable template caching.</td>
</tr>
<tr>
<td>spring.freemarker.charset</td>
<td>UTF-8</td>
<td>Template encoding.</td>
</tr>
<tr>
<td>spring.freemarker.check-template-location</td>
<td>true</td>
<td>Whether to check that the templates location exists.</td>
</tr>
<tr>
<td>spring.freemarker.content-type</td>
<td>text/html</td>
<td>Content-Type value.</td>
</tr>
<tr>
<td>spring.freemarker.enabled</td>
<td>true</td>
<td>Whether to enable MVC view resolution for this technology.</td>
</tr>
<tr>
<td>spring.freemarker.expose-request-attributes</td>
<td>false</td>
<td>Whether all request attributes should be added to the model prior to merging with the template.</td>
</tr>
<tr>
<td>spring.freemarker.expose-session-attributes</td>
<td>false</td>
<td>Whether all HttpSession attributes should be added to the model prior to merging with the template.</td>
</tr>
<tr>
<td>spring.freemarker.expose-spring-macro-helpers</td>
<td>true</td>
<td>Whether to expose a RequestContext for use by Spring's macro library, under the name &quot;springMacroRequestContext&quot;.</td>
</tr>
<tr>
<td>spring.freemarker.prefer-file-system-access</td>
<td>false</td>
<td>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.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.freemarker.prefix</td>
<td></td>
<td>Prefix that gets prepended to view names when building a URL.</td>
</tr>
<tr>
<td>spring.freemarker.request-context-attribute</td>
<td></td>
<td>Name of the RequestContext attribute for all views.</td>
</tr>
<tr>
<td>spring.freemarker.settings.*</td>
<td></td>
<td>Well-known FreeMarker keys which are passed to FreeMarker's Configuration.</td>
</tr>
<tr>
<td>spring.freemarker.suffix</td>
<td>.ftlh</td>
<td>Suffix that gets appended to view names when building a URL.</td>
</tr>
<tr>
<td>spring.freemarker.template-loader-path</td>
<td>[classpath:/templates/]</td>
<td>Comma-separated list of template paths.</td>
</tr>
<tr>
<td>spring.freemarker.view-names</td>
<td></td>
<td>View names that can be resolved.</td>
</tr>
<tr>
<td>spring.groovy.template.allow-request-override</td>
<td>false</td>
<td>Whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.</td>
</tr>
<tr>
<td>spring.groovy.template.allow-session-override</td>
<td>false</td>
<td>Whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.</td>
</tr>
<tr>
<td>spring.groovy.template.cache</td>
<td>false</td>
<td>Whether to enable template caching.</td>
</tr>
<tr>
<td>spring.groovy.template.charset</td>
<td>UTF-8</td>
<td>Template encoding.</td>
</tr>
<tr>
<td>spring.groovy.template.check-template-location</td>
<td>true</td>
<td>Whether to check that the templates location exists.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.auto-escape</code></td>
<td></td>
<td>See GroovyMarkupConfigurer</td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.auto-indent</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.auto-indent-string</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.auto-new-line</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.base-template-class</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.cache-templates</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.declaration-encoding</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.expand-empty-elements</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.locale</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.new-line-string</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.resource-loader-path</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.configuration.use-double-quotes</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>spring.groovy.template.content-type</code></td>
<td><code>text/html</code></td>
<td>Content-Type value.</td>
</tr>
<tr>
<td><code>spring.groovy.template.enabled</code></td>
<td><code>true</code></td>
<td>Whether to enable MVC view resolution for this technology.</td>
</tr>
<tr>
<td><code>spring.groovy.template.expose-request-attributes</code></td>
<td><code>false</code></td>
<td>Whether all request attributes should be added to the model prior to merging with the template.</td>
</tr>
<tr>
<td><code>spring.groovy.template.expose-session-attributes</code></td>
<td><code>false</code></td>
<td>Whether all HttpSession attributes should be added to the model prior to merging with the template.</td>
</tr>
<tr>
<td><code>spring.groovy.template.expose-spring-macro-helpers</code></td>
<td><code>true</code></td>
<td>Whether to expose a RequestContext for use by Spring's macro library, under the name &quot;springMacroRequestContext&quot;.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.groovy.template.prefix</code></td>
<td></td>
<td>Prefix that gets prepended to view names when building a URL.</td>
</tr>
<tr>
<td><code>spring.groovy.template.request-context-attribute</code></td>
<td></td>
<td>Name of the RequestContext attribute for all views.</td>
</tr>
<tr>
<td><code>spring.groovy.template.resource-loader-path</code></td>
<td><code>classpath:/templates/</code></td>
<td>Template path.</td>
</tr>
<tr>
<td><code>spring.groovy.template.suffix</code></td>
<td><code>.tpl</code></td>
<td>Suffix that gets appended to view names when building a URL.</td>
</tr>
<tr>
<td><code>spring.groovy.template.view-names</code></td>
<td></td>
<td>View names that can be resolved.</td>
</tr>
<tr>
<td><code>spring.mustache.allow-request-override</code></td>
<td><code>false</code></td>
<td>Whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.</td>
</tr>
<tr>
<td><code>spring.mustache.allow-session-override</code></td>
<td><code>false</code></td>
<td>Whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.</td>
</tr>
<tr>
<td><code>spring.mustache.cache</code></td>
<td><code>false</code></td>
<td>Whether to enable template caching.</td>
</tr>
<tr>
<td><code>spring.mustache.charset</code></td>
<td><code>UTF-8</code></td>
<td>Template encoding.</td>
</tr>
<tr>
<td><code>spring.mustache.check-template-location</code></td>
<td><code>true</code></td>
<td>Whether to check that the templates location exists.</td>
</tr>
<tr>
<td><code>spring.mustache.content-type</code></td>
<td><code>text/html</code></td>
<td>Content-Type value.</td>
</tr>
<tr>
<td><code>spring.mustache.enabled</code></td>
<td><code>true</code></td>
<td>Whether to enable MVC view resolution for this technology.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.mustache.expose-request-attributes</td>
<td>false</td>
<td>Whether all request attributes should be added to the model prior to merging with the template.</td>
</tr>
<tr>
<td>spring.mustache.expose-session-attributes</td>
<td>false</td>
<td>Whether all HttpSession attributes should be added to the model prior to merging with the template.</td>
</tr>
<tr>
<td>spring.mustache.expose-spring-macro-helpers</td>
<td>true</td>
<td>Whether to expose a RequestContext for use by Spring's macro library, under the name &quot;springMacroRequestContext&quot;.</td>
</tr>
<tr>
<td>spring.mustache.prefix</td>
<td>classpath:/templates/</td>
<td>Prefix to apply to template names.</td>
</tr>
<tr>
<td>spring.mustache.request-context-attribute</td>
<td></td>
<td>Name of the RequestContext attribute for all views.</td>
</tr>
<tr>
<td>spring.mustache.suffix</td>
<td>.mustache</td>
<td>Suffix to apply to template names.</td>
</tr>
<tr>
<td>spring.mustache.view-names</td>
<td></td>
<td>View names that can be resolved.</td>
</tr>
<tr>
<td>spring.thymeleaf.cache</td>
<td>true</td>
<td>Whether to enable template caching.</td>
</tr>
<tr>
<td>spring.thymeleaf.check-template</td>
<td>true</td>
<td>Whether to check that the template exists before rendering it.</td>
</tr>
<tr>
<td>spring.thymeleaf.check-template-location</td>
<td>true</td>
<td>Whether to check that the templates location exists.</td>
</tr>
<tr>
<td>spring.thymeleaf.enable-spring-el-compiler</td>
<td>false</td>
<td>Enable the SpringEL compiler in SpringEL expressions.</td>
</tr>
<tr>
<td>spring.thymeleaf.enabled</td>
<td>true</td>
<td>Whether to enable Thymeleaf view resolution for Web frameworks.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>spring.thymeleaf.encoding</td>
<td>UTF-8</td>
<td>Template files encoding.</td>
</tr>
<tr>
<td>spring.thymeleaf.excluded-view-names</td>
<td></td>
<td>Comma-separated list of view names (patterns allowed) that should be excluded from resolution.</td>
</tr>
<tr>
<td>spring.thymeleaf.mode</td>
<td>HTML</td>
<td>Template mode to be applied to templates. See also Thymeleaf’s TemplateMode enum.</td>
</tr>
<tr>
<td>spring.thymeleaf.prefix</td>
<td>classpath:/templates/</td>
<td>Prefix that gets prepended to view names when building a URL.</td>
</tr>
<tr>
<td>spring.thymeleaf.reactive.chunked-mode-view-names</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>spring.thymeleaf.reactive.full-mode-view-names</td>
<td></td>
<td>Comma-separated list of view names (patterns allowed) that should be executed in FULL mode even if a max chunk size is set.</td>
</tr>
<tr>
<td>spring.thymeleaf.reactive.max-chunk-size</td>
<td>0B</td>
<td>Maximum size of data buffers used for writing to the response. Templates will execute in CHUNKED mode by default if this is set.</td>
</tr>
<tr>
<td>spring.thymeleaf.reactive.media-types</td>
<td></td>
<td>Media types supported by the view technology.</td>
</tr>
<tr>
<td>spring.thymeleaf.render-hidden-markers-before-checkboxes</td>
<td>false</td>
<td>Whether hidden form inputs acting as markers for checkboxes should be rendered before the checkbox element itself.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.thymeleaf.servlet.content-type</td>
<td>text/html</td>
<td>Content-Type value written to HTTP responses.</td>
</tr>
<tr>
<td>spring.thymeleaf.servlet.produce-partial-output-while-processing</td>
<td>true</td>
<td>Whether Thymeleaf should start writing partial output as soon as possible or buffer until template processing is finished.</td>
</tr>
<tr>
<td>spring.thymeleaf.suffix</td>
<td>.html</td>
<td>Suffix that gets appended to view names when building a URL.</td>
</tr>
<tr>
<td>spring.thymeleaf.template-resolver-order</td>
<td></td>
<td>Order of the template resolver in the chain.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>By default, the template resolver is first in the chain.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Order start at 1 and should only be set if you have defined additional &quot;TemplateResolver&quot; beans.</td>
</tr>
<tr>
<td>spring.thymeleaf.view-names</td>
<td></td>
<td>Comma-separated list of view names (patterns allowed) that can be resolved.</td>
</tr>
</tbody>
</table>

**10.A.11. Server Properties**

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server.address</td>
<td></td>
<td>Network address to which the server should bind.</td>
</tr>
<tr>
<td>server.compression.enabled</td>
<td>false</td>
<td>Whether response compression is enabled.</td>
</tr>
<tr>
<td>server.compression.excluded-user-agents</td>
<td></td>
<td>Comma-separated list of user agents for which responses should not be compressed.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>server.compression.mime-types</code></td>
<td><code>[text/html, text/xml, text/plain, text/css, text/javascript, application/javascript, application/json, application/xml]</code></td>
<td>Comma-separated list of MIME types that should be compressed.</td>
</tr>
<tr>
<td><code>server.compression.min-response-size</code></td>
<td><code>2KB</code></td>
<td>Minimum &quot;Content-Length&quot; value that is required for compression to be performed.</td>
</tr>
<tr>
<td><code>server.error.include-binding-errors</code></td>
<td><code>never</code></td>
<td>When to include &quot;errors&quot; attribute.</td>
</tr>
<tr>
<td><code>server.error.include-exception</code></td>
<td><code>false</code></td>
<td>Include the &quot;exception&quot; attribute.</td>
</tr>
<tr>
<td><code>server.error.include-message</code></td>
<td><code>never</code></td>
<td>When to include &quot;message&quot; attribute.</td>
</tr>
<tr>
<td><code>server.error.include-stacktrace</code></td>
<td><code>never</code></td>
<td>When to include the &quot;trace&quot; attribute.</td>
</tr>
<tr>
<td><code>server.error.path</code></td>
<td><code>/error</code></td>
<td>Path of the error controller.</td>
</tr>
<tr>
<td><code>server.error.whitelabel.enabled</code></td>
<td><code>true</code></td>
<td>Whether to enable the default error page displayed in browsers in case of a server error.</td>
</tr>
<tr>
<td><code>server.forward-headers-strategy</code></td>
<td></td>
<td>Strategy for handling X-Forwarded-* headers.</td>
</tr>
<tr>
<td><code>server.http2.enabled</code></td>
<td><code>false</code></td>
<td>Whether to enable HTTP/2 support, if the current environment supports it.</td>
</tr>
<tr>
<td><code>server.jetty.accesslog.append</code></td>
<td><code>false</code></td>
<td>Append to log.</td>
</tr>
<tr>
<td><code>server.jetty.accesslog.custom-format</code></td>
<td></td>
<td>Custom log format, see org.eclipse.jetty.server.CustomRequestLog. If defined, overrides the &quot;format&quot; configuration key.</td>
</tr>
<tr>
<td><code>server.jetty.accesslog.enabled</code></td>
<td><code>false</code></td>
<td>Enable access log.</td>
</tr>
<tr>
<td><code>server.jetty.accesslog.file-date-format</code></td>
<td></td>
<td>Date format to place in log file name.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server.jetty.accesslog.filename</td>
<td></td>
<td>Log filename. If not specified, logs redirect to &quot;System.err&quot;.</td>
</tr>
<tr>
<td>server.jetty.accesslog.format</td>
<td>ncsa</td>
<td>Log format.</td>
</tr>
<tr>
<td>server.jetty.accesslog.ignore-paths</td>
<td></td>
<td>Request paths that should not be logged.</td>
</tr>
<tr>
<td>server.jetty.accesslog.retention-period</td>
<td>31.0</td>
<td>Number of days before rotated log files are deleted.</td>
</tr>
<tr>
<td>server.jetty.connection-idle-timeout</td>
<td></td>
<td>Time that the connection can be idle before it is closed.</td>
</tr>
<tr>
<td>server.jetty.max-http-form-post-size</td>
<td>200000B</td>
<td>Maximum size of the form content in any HTTP post request.</td>
</tr>
<tr>
<td>server.jetty.threads.acceptors</td>
<td>-1.0</td>
<td>Number of acceptor threads to use. When the value is -1, the default, the number of acceptors is derived from the operating environment.</td>
</tr>
<tr>
<td>server.jetty.threads.idle-timeout</td>
<td>60000ms</td>
<td>Maximum thread idle time.</td>
</tr>
<tr>
<td>server.jetty.threads.max</td>
<td>200.0</td>
<td>Maximum number of threads.</td>
</tr>
<tr>
<td>server.jetty.threads.max-queue-capacity</td>
<td></td>
<td>Maximum capacity of the thread pool's backing queue. A default is computed based on the threading configuration.</td>
</tr>
<tr>
<td>server.jetty.threads.min</td>
<td>8.0</td>
<td>Minimum number of threads.</td>
</tr>
<tr>
<td>server.jetty.threads.selectors</td>
<td>-1.0</td>
<td>Number of selector threads to use. When the value is -1, the default, the number of selectors is derived from the operating environment.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server.max-http-header-size</td>
<td>8KB</td>
<td>Maximum size of the HTTP message header.</td>
</tr>
<tr>
<td>server.netty.connection-timeout</td>
<td></td>
<td>Connection timeout of the Netty channel.</td>
</tr>
<tr>
<td>server.netty.h2c-max-content-length</td>
<td>0B</td>
<td>Maximum content length of an H2C upgrade request.</td>
</tr>
<tr>
<td>server.netty.initial-buffer-size</td>
<td>128B</td>
<td>Initial buffer size for HTTP request decoding.</td>
</tr>
<tr>
<td>server.netty.max-chunk-size</td>
<td>8KB</td>
<td>Maximum chunk size that can be decoded for an HTTP request.</td>
</tr>
<tr>
<td>server.netty.max-initial-line-length</td>
<td>4KB</td>
<td>Maximum length that can be decoded for an HTTP request's initial line.</td>
</tr>
<tr>
<td>server.netty.validate-headers</td>
<td>true</td>
<td>Whether to validate headers when decoding requests.</td>
</tr>
<tr>
<td>server.port</td>
<td>8080.0</td>
<td>Server HTTP port.</td>
</tr>
<tr>
<td>server.server-header</td>
<td></td>
<td>Value to use for the Server response header (if empty, no header is sent).</td>
</tr>
<tr>
<td>server.servlet.application-display-name</td>
<td>application</td>
<td>Display name of the application.</td>
</tr>
<tr>
<td>server.servlet.context-parameters.*</td>
<td></td>
<td>Servlet context init parameters.</td>
</tr>
<tr>
<td>server.servlet.context-path</td>
<td></td>
<td>Context path of the application.</td>
</tr>
<tr>
<td>server.servlet.encoding.charset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server.servlet.encoding.enabled</td>
<td>true</td>
<td>Whether to enable http encoding support.</td>
</tr>
<tr>
<td>server.servlet.encoding.force</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server.servlet.encoding.force-request</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server.servlet.encoding.force-response</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server.servlet.encoding.mapping.*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server.servlet.jsp.class-name</td>
<td>org.apache.jasper.servlet.JspServlet</td>
<td>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.</td>
</tr>
<tr>
<td>server.servlet.jsp.init-parameters.*</td>
<td></td>
<td>Init parameters used to configure the JSP servlet.</td>
</tr>
<tr>
<td>server.servlet.jsp.registered</td>
<td>true</td>
<td>Whether the JSP servlet is registered.</td>
</tr>
<tr>
<td>server.servlet.register-default-servlet</td>
<td>false</td>
<td>Whether to register the default Servlet with the container.</td>
</tr>
<tr>
<td>server.servlet.session.cookie.comment</td>
<td></td>
<td>Comment for the session cookie.</td>
</tr>
<tr>
<td>server.servlet.session.cookie.domain</td>
<td></td>
<td>Domain for the session cookie.</td>
</tr>
<tr>
<td>server.servlet.session.cookie.http-only</td>
<td></td>
<td>Whether to use &quot;HttpOnly&quot; cookies for session cookies.</td>
</tr>
<tr>
<td>server.servlet.session.cookie.max-age</td>
<td></td>
<td>Maximum age of the session cookie. If a duration suffix is not specified, seconds will be used.</td>
</tr>
<tr>
<td>server.servlet.session.cookie.name</td>
<td></td>
<td>Session cookie name.</td>
</tr>
<tr>
<td>server.servlet.session.cookie.path</td>
<td></td>
<td>Path of the session cookie.</td>
</tr>
<tr>
<td>server.servlet.session.cookie.secure</td>
<td></td>
<td>Whether to always mark the session cookie as secure.</td>
</tr>
<tr>
<td>server.servlet.session.persistent</td>
<td>false</td>
<td>Whether to persist session data between restarts.</td>
</tr>
<tr>
<td>server.servlet.session.store-dir</td>
<td></td>
<td>Directory used to store session data.</td>
</tr>
<tr>
<td>server.servlet.session.timeout</td>
<td>30m</td>
<td>Session timeout. If a duration suffix is not specified, seconds will be used.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>server.servlet.session.tracking-modes</td>
<td></td>
<td>Session tracking modes.</td>
</tr>
<tr>
<td>server.shutdown</td>
<td>immediate</td>
<td>Type of shutdown that the server will support.</td>
</tr>
<tr>
<td>server.ssl.ciphers</td>
<td></td>
<td>Supported SSL ciphers.</td>
</tr>
<tr>
<td>server.ssl.client-auth</td>
<td></td>
<td>Client authentication mode. Requires a trust store.</td>
</tr>
<tr>
<td>server.ssl.enabled</td>
<td>true</td>
<td>Whether to enable SSL support.</td>
</tr>
<tr>
<td>server.ssl.enabled-protocols</td>
<td></td>
<td>Enabled SSL protocols.</td>
</tr>
<tr>
<td>server.ssl.key-alias</td>
<td></td>
<td>Alias that identifies the key in the key store.</td>
</tr>
<tr>
<td>server.ssl.key-password</td>
<td></td>
<td>Password used to access the key in the key store.</td>
</tr>
<tr>
<td>server.ssl.key-store</td>
<td></td>
<td>Path to the key store that holds the SSL certificate (typically a jks file).</td>
</tr>
<tr>
<td>server.ssl.key-store-password</td>
<td></td>
<td>Password used to access the key store.</td>
</tr>
<tr>
<td>server.ssl.key-store-provider</td>
<td></td>
<td>Provider for the key store.</td>
</tr>
<tr>
<td>server.ssl.key-store-type</td>
<td></td>
<td>Type of the key store.</td>
</tr>
<tr>
<td>server.ssl.protocol</td>
<td>TLS</td>
<td>SSL protocol to use.</td>
</tr>
<tr>
<td>server.ssl.trust-store</td>
<td></td>
<td>Trust store that holds SSL certificates.</td>
</tr>
<tr>
<td>server.ssl.trust-store-password</td>
<td></td>
<td>Password used to access the trust store.</td>
</tr>
<tr>
<td>server.ssl.trust-store-provider</td>
<td></td>
<td>Provider for the trust store.</td>
</tr>
<tr>
<td>server.ssl.trust-store-type</td>
<td></td>
<td>Type of the trust store.</td>
</tr>
<tr>
<td>server.tomcat.accept-count</td>
<td>100.0</td>
<td>Maximum queue length for incoming connection requests when all possible request processing threads are in use.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server.tomcat.accesslog.buffered</td>
<td>true</td>
<td>Whether to buffer output such that it is flushed only periodically.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.check-exists</td>
<td>false</td>
<td>Whether to check for log file existence so it can be recreated if an external process has renamed it.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.condition-if</td>
<td></td>
<td>Whether logging of the request will only be enabled if &quot;ServletRequest.getAttribute(conditionIf)&quot; does not yield null.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.condition-unless</td>
<td></td>
<td>Whether logging of the request will only be enabled if &quot;ServletRequest.getAttribute(conditionUnless)&quot; yield null.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.directory</td>
<td>logs</td>
<td>Directory in which log files are created. Can be absolute or relative to the Tomcat base dir.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.enabled</td>
<td>false</td>
<td>Enable access log.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.encoding</td>
<td></td>
<td>Character set used by the log file. Default to the system default character set.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.file-date-format</td>
<td>.yyyy-MM-dd</td>
<td>Date format to place in the log file name.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.ipv6-canonical</td>
<td>false</td>
<td>Whether to use IPv6 canonical representation format as defined by RFC 5952.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.locale</td>
<td></td>
<td>Locale used to format timestamps in log entries and in log file name suffix. Default to the default locale of the Java process.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server.tomcat.accesslog.max-days</td>
<td>-1.0</td>
<td>Number of days to retain the access log files before they are removed.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.pattern</td>
<td>common</td>
<td>Format pattern for access logs.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.prefix</td>
<td>access_log</td>
<td>Log file name prefix.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.rename-on-rotate</td>
<td>false</td>
<td>Whether to defer inclusion of the date stamp in the file name until rotate time.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.request-attributes-enabled</td>
<td>false</td>
<td>Set request attributes for the IP address, Hostname, protocol, and port used for the request.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.rotate</td>
<td>true</td>
<td>Whether to enable access log rotation.</td>
</tr>
<tr>
<td>server.tomcat.accesslog.suffix</td>
<td>.log</td>
<td>Log file name suffix.</td>
</tr>
<tr>
<td>server.tomcat.additional-tld-skip-patterns</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>server.tomcat.background-processor-delay</td>
<td>10s</td>
<td>Delay between the invocation of backgroundProcess methods. If a duration suffix is not specified, seconds will be used.</td>
</tr>
<tr>
<td>server.tomcat.basedir</td>
<td></td>
<td>Tomcat base directory. If not specified, a temporary directory is used.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>server.tomcat.connection-timeout</code></td>
<td></td>
<td>Amount of time the connector will wait, after accepting a connection, for the request URI line to be presented.</td>
</tr>
<tr>
<td><code>server.tomcat.max-connections</code></td>
<td>8192.0</td>
<td>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 &quot;acceptCount&quot; property.</td>
</tr>
<tr>
<td><code>server.tomcat.max-http-form-post-size</code></td>
<td>2MB</td>
<td>Maximum size of the form content in any HTTP post request.</td>
</tr>
<tr>
<td><code>server.tomcat.max-swallow-size</code></td>
<td>2MB</td>
<td>Maximum amount of request body to swallow.</td>
</tr>
<tr>
<td><code>server.tomcat.mbeanregistry.enabled</code></td>
<td>false</td>
<td>Whether Tomcat's MBean Registry should be enabled.</td>
</tr>
<tr>
<td><code>server.tomcat.processor-cache</code></td>
<td>200.0</td>
<td>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.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>server.tomcat.redirect-context-root</code></td>
<td><code>true</code></td>
<td>Whether requests to the context root should be redirected by appending a <code>/</code> to the path. When using SSL terminated at a proxy, this property should be set to false.</td>
</tr>
<tr>
<td><code>server.tomcat.relaxed-path-vars</code></td>
<td><code>false</code></td>
<td>Comma-separated list of additional unencoded characters that should be allowed in URI paths. Only &quot;&lt; &gt; [ ]^ ` {</td>
</tr>
<tr>
<td><code>server.tomcat.relaxed-query-vars</code></td>
<td><code>false</code></td>
<td>Comma-separated list of additional unencoded characters that should be allowed in URI query strings. Only &quot;&lt; &gt; [ ]^ ` {</td>
</tr>
<tr>
<td><code>server.tomcat.remoteip.host-header</code></td>
<td><code>X-Forwarded-Host</code></td>
<td>Name of the HTTP header from which the remote host is extracted.</td>
</tr>
<tr>
<td><code>server.tomcat.remoteip.internal-proxies</code></td>
<td><code>false</code></td>
<td>Regular expression that matches proxies that are to be trusted.</td>
</tr>
<tr>
<td><code>server.tomcat.remoteip.port-header</code></td>
<td><code>false</code></td>
<td>Name of the HTTP header used to override the original port value.</td>
</tr>
<tr>
<td><code>server.tomcat.remoteip.protocol-header</code></td>
<td><code>false</code></td>
<td>Header that holds the incoming protocol, usually named &quot;X-Forwarded-Proto&quot;.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server.tomcat.remoteip.protocol-header-https-value</td>
<td>https</td>
<td>Value of the protocol header indicating whether the incoming request uses SSL.</td>
</tr>
<tr>
<td>server.tomcat.remoteip.remote-ip-header</td>
<td></td>
<td>Name of the HTTP header from which the remote IP is extracted. For instance, <code>X-FORWARDED-FOR</code>.</td>
</tr>
<tr>
<td>server.tomcat.resource.allow-caching</td>
<td>true</td>
<td>Whether static resource caching is permitted for this web application.</td>
</tr>
<tr>
<td>server.tomcat.resource.cache-ttl</td>
<td></td>
<td>Time-to-live of the static resource cache.</td>
</tr>
<tr>
<td>server.tomcat.threads.max</td>
<td>200.0</td>
<td>Maximum amount of worker threads.</td>
</tr>
<tr>
<td>server.tomcat.threads.min-spare</td>
<td>10.0</td>
<td>Minimum amount of worker threads.</td>
</tr>
<tr>
<td>server.tomcat.uri-encoding</td>
<td>UTF-8</td>
<td>Character encoding to use to decode the URI.</td>
</tr>
<tr>
<td>server.tomcat.use-relative-redirects</td>
<td>false</td>
<td>Whether HTTP 1.1 and later location headers generated by a call to sendRedirect will use relative or absolute redirects.</td>
</tr>
<tr>
<td>server.undertow.accesslog.dir</td>
<td></td>
<td>Undertow access log directory.</td>
</tr>
<tr>
<td>server.undertow.accesslog.enabled</td>
<td>false</td>
<td>Whether to enable the access log.</td>
</tr>
<tr>
<td>server.undertow.accesslog.pattern</td>
<td>common</td>
<td>Format pattern for access logs.</td>
</tr>
<tr>
<td>server.undertow.accesslog.prefix</td>
<td>access_log.</td>
<td>Log file name prefix.</td>
</tr>
<tr>
<td>server.undertow.accesslog.rotate</td>
<td>true</td>
<td>Whether to enable access log rotation.</td>
</tr>
<tr>
<td>server.undertow.accesslog.suffix</td>
<td>log</td>
<td>Log file name suffix.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server.undertow.allow-encoded-slash</td>
<td>false</td>
<td>Whether the server should decode percent encoded slash characters. Enabling encoded slashes can have security implications due to different servers interpreting the slash differently. Only enable this if you have a legacy application that requires it.</td>
</tr>
<tr>
<td>server.undertow.always-set-keep-alive</td>
<td>true</td>
<td>Whether the 'Connection: keep-alive' header should be added to all responses, even if not required by the HTTP specification.</td>
</tr>
<tr>
<td>server.undertow.buffer-size</td>
<td></td>
<td>Size of each buffer. The default is derived from the maximum amount of memory that is available to the JVM.</td>
</tr>
<tr>
<td>server.undertow.decode-url</td>
<td>true</td>
<td>Whether the URL should be decoded. When disabled, percent-encoded characters in the URL will be left as-is.</td>
</tr>
<tr>
<td>server.undertow.direct-buffers</td>
<td></td>
<td>Whether to allocate buffers outside the Java heap. The default is derived from the maximum amount of memory that is available to the JVM.</td>
</tr>
<tr>
<td>server.undertow.eager-filter-init</td>
<td>true</td>
<td>Whether servlet filters should be initialized on startup.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server.undertow.max-cookies</td>
<td>200.0</td>
<td>Maximum number of cookies that are allowed. This limit exists to prevent hash collision based DOS attacks.</td>
</tr>
<tr>
<td>server.undertow.max-headers</td>
<td></td>
<td>Maximum number of headers that are allowed. This limit exists to prevent hash collision based DOS attacks.</td>
</tr>
<tr>
<td>server.undertow.max-http-post-size</td>
<td>-1B</td>
<td>Maximum size of the HTTP post content. When the value is -1, the default, the size is unlimited.</td>
</tr>
<tr>
<td>server.undertow.max-parameters</td>
<td></td>
<td>Maximum number of query or path parameters that are allowed. This limit exists to prevent hash collision based DOS attacks.</td>
</tr>
<tr>
<td>server.undertow.no-request-timeout</td>
<td></td>
<td>Amount of time a connection can sit idle without processing a request, before it is closed by the server.</td>
</tr>
<tr>
<td>server.undertow.options.server.*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server.undertow.options.socket.*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server.undertow.preserve-path-on-forward</td>
<td>false</td>
<td>Whether to preserve the path of a request when it is forwarded.</td>
</tr>
<tr>
<td>server.undertow.threads.io</td>
<td></td>
<td>Number of I/O threads to create for the worker. The default is derived from the number of available processors.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server.undertow.threads.worker</td>
<td></td>
<td>Number of worker threads. The default is 8 times the number of I/O threads.</td>
</tr>
<tr>
<td>server.undertow.url-charset</td>
<td>UTF-8</td>
<td>Charset used to decode URLs.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.security.filter.order</td>
<td>-100.0</td>
<td>Security filter chain order.</td>
</tr>
<tr>
<td>spring.security.oauth2.client.provider.*</td>
<td></td>
<td>OAuth provider details.</td>
</tr>
<tr>
<td>spring.security.oauth2.client.registration.*</td>
<td></td>
<td>OAuth client registrations.</td>
</tr>
<tr>
<td>spring.security.oauth2.resourceserver.jwt.issuer-uri</td>
<td></td>
<td>URI that can either be an OpenID Connect discovery endpoint or an OAuth 2.0 Authorization Server Metadata endpoint defined by RFC 8414.</td>
</tr>
<tr>
<td>spring.security.oauth2.resourceserver.jwt.jwks-uri</td>
<td></td>
<td>JSON Web Key URI to use to verify the JWT token.</td>
</tr>
<tr>
<td>spring.security.oauth2.resourceserver.jwt.jws-algorithm</td>
<td>RS256</td>
<td>JSON Web Algorithm used for verifying the digital signatures.</td>
</tr>
<tr>
<td>spring.security.oauth2.resourceserver.jwt.public-key-location</td>
<td></td>
<td>Location of the file containing the public key used to verify a JWT.</td>
</tr>
<tr>
<td>spring.security.oauth2.resourceserver.oauth2.token.client-id</td>
<td></td>
<td>Client id used to authenticate with the token introspection endpoint.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.security.oauth2.resourceserver.opaque-token.client-secret</td>
<td></td>
<td>Client secret used to authenticate with the token introspection endpoint.</td>
</tr>
<tr>
<td>spring.security.oauth2.resourceserver.opaque-token.introspection-uri</td>
<td></td>
<td>OAuth 2.0 endpoint through which token introspection is accomplished.</td>
</tr>
<tr>
<td>spring.security.saml2.relyingparty.registration.*</td>
<td></td>
<td>SAML2 relying party registrations.</td>
</tr>
<tr>
<td>spring.security.user.name</td>
<td>user</td>
<td>Default user name.</td>
</tr>
<tr>
<td>spring.security.user.password</td>
<td></td>
<td>Password for the default user name.</td>
</tr>
<tr>
<td>spring.security.user.roles</td>
<td></td>
<td>Granted roles for the default user name.</td>
</tr>
</tbody>
</table>

### 10.A.13. RSocket Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.rsocket.server.address</td>
<td></td>
<td>Network address to which the server should bind.</td>
</tr>
<tr>
<td>spring.rsocket.server.fragment-size</td>
<td></td>
<td>Maximum transmission unit. Frames larger than the specified value are fragmented.</td>
</tr>
<tr>
<td>spring.rsocket.server.mapping-path</td>
<td></td>
<td>Path under which RSocket handles requests (only works with websocket transport).</td>
</tr>
<tr>
<td>spring.rsocket.server.port</td>
<td></td>
<td>Server port.</td>
</tr>
<tr>
<td>spring.rsocket.server.ssl.ciphers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.rsocket.server.ssl.client-auth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.rsocket.server.ssl.enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.rsocket.server.ssl.enabled-protocols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.rsocket.server.ssl.key-alias</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.rsocket.server.ssl.key-password</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.rsocket.server.ssl.key-store</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>management.auditevents.enabled</td>
<td>true</td>
<td>Whether to enable storage of audit events.</td>
</tr>
<tr>
<td>management.cloudfoundry.enabled</td>
<td>true</td>
<td>Whether to enable extended Cloud Foundry actuator endpoints.</td>
</tr>
<tr>
<td>management.cloudfoundry.skip-ssl-validation</td>
<td>false</td>
<td>Whether to skip SSL verification for Cloud Foundry actuator endpoint security calls.</td>
</tr>
<tr>
<td>management.endpoint.auditevents.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.auditevents.enabled</td>
<td>true</td>
<td>Whether to enable the auditevents endpoint.</td>
</tr>
<tr>
<td>management.endpoint.beans.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.beans.enabled</td>
<td>true</td>
<td>Whether to enable the beans endpoint.</td>
</tr>
<tr>
<td>management.endpoint.caches.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.caches.enabled</td>
<td>true</td>
<td>Whether to enable the caches endpoint.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.endpoint.conditions.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.conditions.enabled</td>
<td>true</td>
<td>Whether to enable the conditions endpoint.</td>
</tr>
<tr>
<td>management.endpoint.configprops.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.configprops.enabled</td>
<td>true</td>
<td>Whether to enable the configprops endpoint.</td>
</tr>
<tr>
<td>management.endpoint.configprops.keys-to-sanitize</td>
<td>[password, secret, key, token, .<em>credentials.</em>, vcap_services, sun.java.command]</td>
<td>Keys that should be sanitized. Keys can be simple strings that the property ends with or regular expressions.</td>
</tr>
<tr>
<td>management.endpoint.env.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.env.enabled</td>
<td>true</td>
<td>Whether to enable the env endpoint.</td>
</tr>
<tr>
<td>management.endpoint.env.keys-to-sanitize</td>
<td>[password, secret, key, token, .<em>credentials.</em>, vcap_services, sun.java.command]</td>
<td>Keys that should be sanitized. Keys can be simple strings that the property ends with or regular expressions.</td>
</tr>
<tr>
<td>management.endpoint.flyway.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.flyway.enabled</td>
<td>true</td>
<td>Whether to enable the flyway endpoint.</td>
</tr>
<tr>
<td>management.endpoint.health.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.health.enabled</td>
<td>true</td>
<td>Whether to enable the health endpoint.</td>
</tr>
<tr>
<td>management.endpoint.health.group.*</td>
<td></td>
<td>Health endpoint groups.</td>
</tr>
<tr>
<td>management.endpoint.health.probes.enabled</td>
<td>false</td>
<td>Whether to enable liveness and readiness probes.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.endpoint.health.roles</td>
<td></td>
<td>Roles used to determine whether or not a user is authorized to be shown details. When empty, all authenticated users are authorized.</td>
</tr>
<tr>
<td>management.endpoint.health.show-components</td>
<td></td>
<td>When to show components. If not specified the 'show-details' setting will be used.</td>
</tr>
<tr>
<td>management.endpoint.health.show-details</td>
<td>never</td>
<td>When to show full health details.</td>
</tr>
<tr>
<td>management.endpoint.health.status.http-mapping.*</td>
<td></td>
<td>Mapping of health statuses to HTTP status codes. By default, registered health statuses map to sensible defaults (for example, UP maps to 200).</td>
</tr>
<tr>
<td>management.endpoint.health.status.order</td>
<td>[DOWN, OUT_OF_SERVICE, UP, UNKNOWN]</td>
<td>Comma-separated list of health statuses in order of severity.</td>
</tr>
<tr>
<td>management.endpoint.heapdump.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.heapdump.enabled</td>
<td>true</td>
<td>Whether to enable the heapdump endpoint.</td>
</tr>
<tr>
<td>management.endpoint.httptrace.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.httptrace.enabled</td>
<td>true</td>
<td>Whether to enable the httptrace endpoint.</td>
</tr>
<tr>
<td>management.endpoint.info.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.info.enabled</td>
<td>true</td>
<td>Whether to enable the info endpoint.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>---------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>management.endpoint.integrationgraph.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.integrationgraph.enabled</td>
<td>true</td>
<td>Whether to enable the integrationgraph endpoint.</td>
</tr>
<tr>
<td>management.endpoint.jolokia.config.*</td>
<td></td>
<td>Jolokia settings. Refer to the documentation of Jolokia for more details.</td>
</tr>
<tr>
<td>management.endpoint.jolokia.enabled</td>
<td>true</td>
<td>Whether to enable the jolokia endpoint.</td>
</tr>
<tr>
<td>management.endpoint.liquibase.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.liquibase.enabled</td>
<td>true</td>
<td>Whether to enable the liquibase endpoint.</td>
</tr>
<tr>
<td>management.endpoint.logfile.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.logfile.enabled</td>
<td>true</td>
<td>Whether to enable the logfile endpoint.</td>
</tr>
<tr>
<td>management.endpoint.logfile.external-file</td>
<td></td>
<td>External Logfile to be accessed. Can be used if the logfile is written by output redirect and not by the logging system itself.</td>
</tr>
<tr>
<td>management.endpoint.loggers.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.loggers.enabled</td>
<td>true</td>
<td>Whether to enable the loggers endpoint.</td>
</tr>
<tr>
<td>management.endpoint.mappings.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.mappings.enabled</td>
<td>true</td>
<td>Whether to enable the mappings endpoint.</td>
</tr>
<tr>
<td>management.endpoint.metrics.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>management.endpoint.metrics.enabled</td>
<td>true</td>
<td>Whether to enable the metrics endpoint.</td>
</tr>
<tr>
<td>management.endpoint.prometheus.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.prometheus.enabled</td>
<td>true</td>
<td>Whether to enable the prometheus endpoint.</td>
</tr>
<tr>
<td>management.endpoint.scheduledtasks.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.scheduledtasks.enabled</td>
<td>true</td>
<td>Whether to enable the scheduledtasks endpoint.</td>
</tr>
<tr>
<td>management.endpoint.sessions.enabled</td>
<td>true</td>
<td>Whether to enable the sessions endpoint.</td>
</tr>
<tr>
<td>management.endpoint.shutdown.enabled</td>
<td>false</td>
<td>Whether to enable the shutdown endpoint.</td>
</tr>
<tr>
<td>management.endpoint.startup.enabled</td>
<td>true</td>
<td>Whether to enable the startup endpoint.</td>
</tr>
<tr>
<td>management.endpoint.threaddump.cache.time-to-live</td>
<td>0ms</td>
<td>Maximum time that a response can be cached.</td>
</tr>
<tr>
<td>management.endpoint.threaddump.enabled</td>
<td>true</td>
<td>Whether to enable the threaddump endpoint.</td>
</tr>
<tr>
<td>management.endpoints.enabled-by-default</td>
<td></td>
<td>Whether to enable or disable all endpoints by default.</td>
</tr>
<tr>
<td>management.endpoints.jmx.domain</td>
<td>org.springframework.boot</td>
<td>Endpoints JMX domain name. Fallback to 'spring.jmx.default-domain' if set.</td>
</tr>
<tr>
<td>management.endpoints.jmx.exposure.exclude</td>
<td></td>
<td>Endpoint IDs that should be excluded or '*' for all.</td>
</tr>
<tr>
<td>management.endpoints.jmx.exposure.include</td>
<td>*</td>
<td>Endpoint IDs that should be included or '*' for all.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.endpoints.jmx.static-names</td>
<td></td>
<td>Additional static properties to append to all ObjectNames of MBeans representing Endpoints.</td>
</tr>
<tr>
<td>management.endpoints.migrate-legacy-ids</td>
<td>false</td>
<td>Whether to transparently migrate legacy endpoint IDs.</td>
</tr>
<tr>
<td>management.endpoints.web.base-path</td>
<td>/actuator</td>
<td>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.</td>
</tr>
<tr>
<td>management.endpoints.web.cors.allow-credentials</td>
<td></td>
<td>Whether credentials are supported. When not set, credentials are not supported.</td>
</tr>
<tr>
<td>management.endpoints.web.cors.allowed-headers</td>
<td></td>
<td>Comma-separated list of headers to allow in a request. ‘*’ allows all headers.</td>
</tr>
<tr>
<td>management.endpoints.web.cors.allowed-methods</td>
<td></td>
<td>Comma-separated list of methods to allow. ‘*’ allows all methods. When not set, defaults to GET.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.endpoints.web.cors.allowed-origins</td>
<td></td>
<td>Comma-separated list of origins to allow. &quot;*&quot; allows all origins. When not set, CORS support is disabled.</td>
</tr>
<tr>
<td>management.endpoints.web.cors.exposed-headers</td>
<td></td>
<td>Comma-separated list of headers to include in a response.</td>
</tr>
<tr>
<td>management.endpoints.web.cors.max-age</td>
<td>1800s</td>
<td>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.</td>
</tr>
<tr>
<td>management.endpoints.web.exposure.exclude</td>
<td></td>
<td>Endpoint IDs that should be excluded or &quot;*&quot; for all.</td>
</tr>
<tr>
<td>management.endpoints.web.exposure.include</td>
<td>[health, info]</td>
<td>Endpoint IDs that should be included or &quot;*&quot; for all.</td>
</tr>
<tr>
<td>management.endpoints.web.path-mapping.*</td>
<td></td>
<td>Mapping between endpoint IDs and the path that should expose them.</td>
</tr>
<tr>
<td>management.health.cassandra.enabled</td>
<td>true</td>
<td>Whether to enable Cassandra health check.</td>
</tr>
<tr>
<td>management.health.couchbase.enabled</td>
<td>true</td>
<td>Whether to enable Couchbase health check.</td>
</tr>
<tr>
<td>management.health.db.enabled</td>
<td>true</td>
<td>Whether to enable database health check.</td>
</tr>
<tr>
<td>management.health.db.ignore-routing-data-sources</td>
<td>false</td>
<td>Whether to ignore AbstractRoutingDataSources when creating database health indicators.</td>
</tr>
<tr>
<td>management.health.defaults.enabled</td>
<td>true</td>
<td>Whether to enable default health indicators.</td>
</tr>
<tr>
<td>management.health.diskspace.enabled</td>
<td>true</td>
<td>Whether to enable disk space health check.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.health.diskspace.path</td>
<td></td>
<td>Path used to compute the available disk space.</td>
</tr>
<tr>
<td>management.health.diskspace.threshold</td>
<td>10MB</td>
<td>Minimum disk space that should be available.</td>
</tr>
<tr>
<td>management.health.elasticsearch.enabled</td>
<td>true</td>
<td>Whether to enable Elasticsearch health check.</td>
</tr>
<tr>
<td>management.health.influxdb.enabled</td>
<td>true</td>
<td>Whether to enable InfluxDB health check.</td>
</tr>
<tr>
<td>management.health.jms.enabled</td>
<td>true</td>
<td>Whether to enable JMS health check.</td>
</tr>
<tr>
<td>management.health.ldap.enabled</td>
<td>true</td>
<td>Whether to enable LDAP health check.</td>
</tr>
<tr>
<td>management.health.livenessstate.enabled</td>
<td>false</td>
<td>Whether to enable liveness state health check.</td>
</tr>
<tr>
<td>management.health.mail.enabled</td>
<td>true</td>
<td>Whether to enable Mail health check.</td>
</tr>
<tr>
<td>management.health.mongo.enabled</td>
<td>true</td>
<td>Whether to enable MongoDB health check.</td>
</tr>
<tr>
<td>management.health.neo4j.enabled</td>
<td>true</td>
<td>Whether to enable Neo4j health check.</td>
</tr>
<tr>
<td>management.health.ping.enabled</td>
<td>true</td>
<td>Whether to enable ping health check.</td>
</tr>
<tr>
<td>management.health.rabbit.enabled</td>
<td>true</td>
<td>Whether to enable RabbitMQ health check.</td>
</tr>
<tr>
<td>management.health.readinessstate.enabled</td>
<td>false</td>
<td>Whether to enable readiness state health check.</td>
</tr>
<tr>
<td>management.health.redis.enabled</td>
<td>true</td>
<td>Whether to enable Redis health check.</td>
</tr>
<tr>
<td>management.health.solr.enabled</td>
<td>true</td>
<td>Whether to enable Solr health check.</td>
</tr>
<tr>
<td>management.health.status.order</td>
<td>[DOWN, OUT_OF_SERVICE, UP, UNKNOWN]</td>
<td>Path used to compute the available disk space.</td>
</tr>
<tr>
<td>management.info.build.enabled</td>
<td>true</td>
<td>Whether to enable build info.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.info.defaults.enabled</td>
<td>true</td>
<td>Whether to enable default info contributors.</td>
</tr>
<tr>
<td>management.info.env.enabled</td>
<td>true</td>
<td>Whether to enable environment info.</td>
</tr>
<tr>
<td>management.info.git.enabled</td>
<td>true</td>
<td>Whether to enable git info.</td>
</tr>
<tr>
<td>management.info.git.mode</td>
<td>simple</td>
<td>Mode to use to expose git information.</td>
</tr>
<tr>
<td>management.metrics.distribution.maximum-expected-value.*</td>
<td></td>
<td>Maximum value that meter IDs starting with the specified name are expected to observe. The longest match wins. Values can be specified as a long or as a Duration value (for timer meters, defaulting to ms if no unit specified).</td>
</tr>
<tr>
<td>management.metrics.distribution.minimum-expected-value.*</td>
<td></td>
<td>Minimum value that meter IDs starting with the specified name are expected to observe. The longest match wins. Values can be specified as a long or as a Duration value (for timer meters, defaulting to ms if no unit specified).</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>management.metrics.distribution.percentiles-histogram.*</code></td>
<td></td>
<td>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 <code>all</code> can also be used to configure all meters.</td>
</tr>
<tr>
<td><code>management.metrics.distribution.percentiles.*</code></td>
<td></td>
<td>Specific computed non-aggregable percentiles to ship to the backend for meter IDs starting with the specified name. The longest match wins, the key <code>all</code> can also be used to configure all meters.</td>
</tr>
<tr>
<td><code>management.metrics.distribution.slo.*</code></td>
<td></td>
<td>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 long or as a Duration value (for timer meters, defaulting to ms if no unit specified).</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.enable.*</td>
<td></td>
<td>Whether meter IDs starting with the specified name should be enabled. The longest match wins, the key <code>all</code> can also be used to configure all meters.</td>
</tr>
<tr>
<td>management.metrics.export.appoptics.api-token</td>
<td></td>
<td>AppOptics API token.</td>
</tr>
<tr>
<td>management.metrics.export.appoptics.batch-size</td>
<td>500.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>management.metrics.export.appoptics.connect-timeout</td>
<td>5s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.appoptics.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.appoptics.floor-times</td>
<td>false</td>
<td>Whether to ship a floored time, useful when sending measurements from multiple hosts to align them on a given time boundary.</td>
</tr>
<tr>
<td>management.metrics.export.appoptics.host-tag</td>
<td>instance</td>
<td>Tag that will be mapped to &quot;@host&quot; when shipping metrics to AppOptics.</td>
</tr>
<tr>
<td>management.metrics.export.appoptics.read-timeout</td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.appoptics.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.appoptics.uri</td>
<td><a href="https://api.appoptics.com/v1/measurements">https://api.appoptics.com/v1/measurements</a></td>
<td>URI to ship metrics to.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>management.metrics.export.atlas.batch-size</code></td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td><code>management.metrics.export.atlas.config-refresh-frequency</code></td>
<td>10s</td>
<td>Frequency for refreshing config settings from the LWC service.</td>
</tr>
<tr>
<td><code>management.metrics.export.atlas.config-time-to-live</code></td>
<td>150s</td>
<td>Time to live for subscriptions from the LWC service.</td>
</tr>
<tr>
<td><code>management.metrics.export.atlas.connect-timeout</code></td>
<td>1s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td><code>management.metrics.export.atlas.enabled</code></td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td><code>management.metrics.export.atlas.lwc-enabled</code></td>
<td>false</td>
<td>Whether to enable streaming to Atlas LWC.</td>
</tr>
<tr>
<td><code>management.metrics.export.atlas.meter-time-to-live</code></td>
<td>15m</td>
<td>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.</td>
</tr>
<tr>
<td><code>management.metrics.export.atlas.num-threads</code></td>
<td>4.0</td>
<td>Number of threads to use with the metrics publishing scheduler.</td>
</tr>
<tr>
<td><code>management.metrics.export.atlas.read-timeout</code></td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.export.atlas.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.datadog.api-key</td>
<td></td>
<td>Datadog API key.</td>
</tr>
<tr>
<td>management.metrics.export.datadog.application-key</td>
<td></td>
<td>Datadog application key. Not strictly required, but improves the Datadog experience by sending meter descriptions, types, and base units to Datadog.</td>
</tr>
<tr>
<td>management.metrics.export.datadog.batch-size</td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>management.metrics.export.datadog.connect-timeout</td>
<td>1s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.datadog.descriptions</td>
<td>true</td>
<td>Whether to publish descriptions metadata to Datadog. Turn this off to minimize the amount of metadata sent.</td>
</tr>
<tr>
<td>management.metrics.export.datadog.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.datadog.host-tag</td>
<td>instance</td>
<td>Tag that will be mapped to &quot;host&quot; when shipping metrics to Datadog.</td>
</tr>
<tr>
<td>management.metrics.export.datadog.read-timeout</td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.datadog.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.export.datadog.uri</td>
<td><a href="https://api.datadoghq.com">https://api.datadoghq.com</a></td>
<td>URI to ship metrics to. If you need to publish metrics to an internal proxy en-route to Datadog, you can define the location of the proxy with this.</td>
</tr>
<tr>
<td>management.metrics.export.defaults.enabled</td>
<td>true</td>
<td>Whether to enable default metrics exporters.</td>
</tr>
<tr>
<td>management.metrics.export.dynatrace.api-token</td>
<td></td>
<td>Dynatrace authentication token.</td>
</tr>
<tr>
<td>management.metrics.export.dynatrace.batch-size</td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>management.metrics.export.dynatrace.connect-timeout</td>
<td>1s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.dynatrace.device-id</td>
<td></td>
<td>ID of the custom device that is exporting metrics to Dynatrace.</td>
</tr>
<tr>
<td>management.metrics.export.dynatrace.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.dynatrace.group</td>
<td></td>
<td>Group for exported metrics. Used to specify custom device group name in the Dynatrace UI.</td>
</tr>
<tr>
<td>management.metrics.export.dynatrace.read-timeout</td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.dynatrace.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>management.metrics.export.dynatrace.technology-type</code></td>
<td><code>java</code></td>
<td>Technology type for exported metrics. Used to group metrics under a logical technology name in the Dynatrace UI.</td>
</tr>
<tr>
<td><code>management.metrics.export.dynatrace.uri</code></td>
<td></td>
<td>URI to ship metrics to. Should be used for SaaS, self managed instances or to en-route through an internal proxy.</td>
</tr>
<tr>
<td><code>management.metrics.export.elastic.auto-create-index</code></td>
<td><code>true</code></td>
<td>Whether to create the index automatically if it does not exist.</td>
</tr>
<tr>
<td><code>management.metrics.export.elastic.batch-size</code></td>
<td><code>10000.0</code></td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td><code>management.metrics.export.elastic.connect-timeout</code></td>
<td><code>1s</code></td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td><code>management.metrics.export.elastic.enabled</code></td>
<td><code>true</code></td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td><code>management.metrics.export.elastic.host</code></td>
<td><code>http://localhost:9200</code></td>
<td>Host to export metrics to.</td>
</tr>
<tr>
<td><code>management.metrics.export.elastic.index</code></td>
<td><code>micrometer-metrics</code></td>
<td>Index to export metrics to.</td>
</tr>
<tr>
<td><code>management.metrics.export.elastic.index-date-format</code></td>
<td><code>yyyy-MM</code></td>
<td>Index date format used for rolling indices. Appended to the index name.</td>
</tr>
<tr>
<td><code>management.metrics.export.elastic.index-date-separator</code></td>
<td><code>-</code></td>
<td>Prefix to separate the index name from the date format used for rolling indices.</td>
</tr>
<tr>
<td><code>management.metrics.export.elastic.password</code></td>
<td></td>
<td>Login password of the Elastic server.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.export.elastic.pipeline</td>
<td></td>
<td>Ingest pipeline name. By default, events are not pre-processed.</td>
</tr>
<tr>
<td>management.metrics.export.elastic.read-timeout</td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.elastic.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.elastic.timestamp-field-name</td>
<td>@timestamp</td>
<td>Name of the timestamp field.</td>
</tr>
<tr>
<td>management.metrics.export.elastic.user-name</td>
<td></td>
<td>Login user of the Elastic server.</td>
</tr>
<tr>
<td>management.metrics.export.ganglia.addressing-mode</td>
<td>multicast</td>
<td>UDP addressing mode, either unicast or multicast.</td>
</tr>
<tr>
<td>management.metrics.export.ganglia.duration-units</td>
<td>milliseconds</td>
<td>Base time unit used to report durations.</td>
</tr>
<tr>
<td>management.metrics.export.ganglia.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to Ganglia is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.ganglia.host</td>
<td>localhost</td>
<td>Host of the Ganglia server to receive exported metrics.</td>
</tr>
<tr>
<td>management.metrics.export.ganglia.port</td>
<td>8649.0</td>
<td>Port of the Ganglia server to receive exported metrics.</td>
</tr>
<tr>
<td>management.metrics.export.ganglia.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.ganglia.time-to-live</td>
<td>1.0</td>
<td>Time to live for metrics on Ganglia. Set the multi-cast Time-To-Live to be one greater than the number of hops (routers) between the hosts.</td>
</tr>
<tr>
<td>management.metrics.export.graphite.duration-units</td>
<td>milliseconds</td>
<td>Base time unit used to report durations.</td>
</tr>
<tr>
<td>management.metrics.export.graphite.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to Graphite is enabled.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.export.graphite.graphite-tags-enabled</td>
<td></td>
<td>Whether Graphite tags should be used, as opposed to a hierarchical naming convention. Enabled by default unless &quot;tagsAsPrefix&quot; is set.</td>
</tr>
<tr>
<td>management.metrics.export.graphite.host</td>
<td>localhost</td>
<td>Host of the Graphite server to receive exported metrics.</td>
</tr>
<tr>
<td>management.metrics.export.graphite.port</td>
<td>2004.0</td>
<td>Port of the Graphite server to receive exported metrics.</td>
</tr>
<tr>
<td>management.metrics.export.graphite.protocol</td>
<td>pickled</td>
<td>Protocol to use while shipping data to Graphite.</td>
</tr>
<tr>
<td>management.metrics.export.graphite.rate-units</td>
<td>seconds</td>
<td>Base time unit used to report rates.</td>
</tr>
<tr>
<td>management.metrics.export.graphite.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.graphite.tags-as-prefix</td>
<td>[]</td>
<td>For the hierarchical naming convention, turn the specified tag keys into part of the metric prefix. Ignored if &quot;graphiteTagsEnabled&quot; is true.</td>
</tr>
<tr>
<td>management.metrics.export.humio.api-token</td>
<td></td>
<td>Humio API token.</td>
</tr>
<tr>
<td>management.metrics.export.humio.batch-size</td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>management.metrics.export.humio.connect-timeout</td>
<td>5s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.humio.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.export.humio.read-timeout</td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.humio.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.humio.tags.*</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>management.metrics.export.humio.uri</td>
<td><a href="https://cloud.humio.com">https://cloud.humio.com</a></td>
<td>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.</td>
</tr>
<tr>
<td>management.metrics.export.influx.auto-create-db</td>
<td>true</td>
<td>Whether to create the Influx database if it does not exist before attempting to publish metrics to it.</td>
</tr>
<tr>
<td>management.metrics.export.influx.batch-size</td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>management.metrics.export.influx.compressed</td>
<td>true</td>
<td>Whether to enable GZIP compression of metrics batches published to Influx.</td>
</tr>
<tr>
<td>management.metrics.export.influx.connect-timeout</td>
<td>1s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.influx.consistency</td>
<td>one</td>
<td>Write consistency for each point.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.db</code></td>
<td><code>mydb</code></td>
<td>Tag that will be mapped to &quot;host&quot; when shipping metrics to Influx.</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.enabled</code></td>
<td><code>true</code></td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.password</code></td>
<td></td>
<td>Login password of the Influx server.</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.read-timeout</code></td>
<td><code>10s</code></td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.retention-duration</code></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.retention-policy</code></td>
<td></td>
<td>Retention policy to use (Influx writes to the DEFAULT retention policy if one is not specified).</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.retention-replication-factor</code></td>
<td></td>
<td>How many copies of the data are stored in the cluster. Must be 1 for a single node instance.</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.retention-shard-duration</code></td>
<td></td>
<td>Time range covered by a shard group. For instance 2w, check the influx documentation for more details on the duration format.</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.step</code></td>
<td><code>1m</code></td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td><code>management.metrics.export.influx.user-name</code></td>
<td></td>
<td>Login user of the Influx server.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>management.metrics.export.jmx.domain</td>
<td>metrics</td>
<td>Metrics JMX domain name.</td>
</tr>
<tr>
<td>management.metrics.export.jmx.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to JMX is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.jmx.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.kairos.batch-size</td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>management.metrics.export.kairos.connect-timeout</td>
<td>1s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.kairos.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.kairos.password</td>
<td></td>
<td>Login password of the KairosDB server.</td>
</tr>
<tr>
<td>management.metrics.export.kairos.read-timeout</td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.kairos.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.kairos.user-name</td>
<td></td>
<td>Login user of the KairosDB server.</td>
</tr>
<tr>
<td>management.metrics.export.newrelic.account-id</td>
<td></td>
<td>New Relic account ID.</td>
</tr>
<tr>
<td>management.metrics.export.newrelic.api-key</td>
<td></td>
<td>New Relic API key.</td>
</tr>
<tr>
<td>management.metrics.export.newrelic.batch-size</td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>management.metrics.export.newrelic.client-provider-type</code></td>
<td></td>
<td>Client provider type to use.</td>
</tr>
<tr>
<td><code>management.metrics.export.newrelic.connect-timeout</code></td>
<td>1s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td><code>management.metrics.export.newrelic.enabled</code></td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td><code>management.metrics.export.newrelic.event-type</code></td>
<td>SpringBootSample</td>
<td>The event type that should be published. This property will be ignored if 'meter-name-event-type-enabled' is set to 'true'.</td>
</tr>
<tr>
<td><code>management.metrics.export.newrelic.meter-name-event-type-enabled</code></td>
<td>false</td>
<td>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.</td>
</tr>
<tr>
<td><code>management.metrics.export.newrelic.read-timeout</code></td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td><code>management.metrics.export.newrelic.step</code></td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td><code>management.metrics.export.newrelic.uri</code></td>
<td><a href="https://insights-collector.newrelic.com">https://insights-collector.newrelic.com</a></td>
<td>URI to ship metrics to.</td>
</tr>
<tr>
<td><code>management.metrics.export.prometheus.descriptions</code></td>
<td>true</td>
<td>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.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to Prometheus is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.histogram-flavor</td>
<td>prometheus</td>
<td>Histogram type for backing DistributionSummary and Timer.</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.pushgateway.enabled</td>
<td>false</td>
<td>Enable publishing via a Prometheus Pushgateway.</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.pushgateway.grouping-key.*</td>
<td></td>
<td>Grouping key for the pushed metrics.</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.pushgateway.job</td>
<td></td>
<td>Job identifier for this application instance.</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.pushgateway.password</td>
<td></td>
<td>Login password of the Prometheus Pushgateway.</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.pushgateway.push-rate</td>
<td>1m</td>
<td>Frequency with which to push metrics.</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.pushgateway.shutdown-operation</td>
<td>none</td>
<td>Operation that should be performed on shutdown.</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.pushgateway.username</td>
<td></td>
<td>Login user of the Prometheus Pushgateway.</td>
</tr>
<tr>
<td>management.metrics.export.prometheus.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.signalfx.access-token</td>
<td></td>
<td>SignalFX access token.</td>
</tr>
<tr>
<td>management.metrics.export.signalfx.batch-size</td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more</td>
</tr>
<tr>
<td></td>
<td></td>
<td>measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>management.metrics.export.signalfx.connect-timeout</td>
<td>1s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.export.signalfx.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.signalfx.read-timeout</td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.signalfx.source</td>
<td></td>
<td>Uniquely identifies the app instance that is publishing metrics to SignalFx. Defaults to the local host name.</td>
</tr>
<tr>
<td>management.metrics.export.signalfx.step</td>
<td>10s</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.signalfx.uri</td>
<td><a href="https://ingest.signalfx.com">https://ingest.signalfx.com</a></td>
<td>URI to ship metrics to.</td>
</tr>
<tr>
<td>management.metrics.export.simple.enabled</td>
<td>true</td>
<td>Whether, in the absence of any other exporter, exporting of metrics to an in-memory backend is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.simple.mode</td>
<td>cumulative</td>
<td>Counting mode.</td>
</tr>
<tr>
<td>management.metrics.export.simple.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.stackdriver.batch-size</td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>management.metrics.export.stackdriver.connect-timeout</td>
<td>1s</td>
<td>Connection timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.stackdriver.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.stackdriver.project-id</td>
<td></td>
<td>Identifier of the Google Cloud project to monitor.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.export.stackdriver.read-timeout</td>
<td>10s</td>
<td>Read timeout for requests to this backend.</td>
</tr>
<tr>
<td>management.metrics.export.stackdriver.resource-type</td>
<td>global</td>
<td>Monitored resource type.</td>
</tr>
<tr>
<td>management.metrics.export.stackdriver.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.statsd.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to StatsD is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.statsd.flavor</td>
<td>datadog</td>
<td>StatsD line protocol to use.</td>
</tr>
<tr>
<td>management.metrics.export.statsd.host</td>
<td>localhost</td>
<td>Host of the StatsD server to receive exported metrics.</td>
</tr>
<tr>
<td>management.metrics.export.statsd.max-packet-length</td>
<td>1400.0</td>
<td>Total length of a single payload should be kept within your network's MTU.</td>
</tr>
<tr>
<td>management.metrics.export.statsd.polling-frequency</td>
<td>10s</td>
<td>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.</td>
</tr>
<tr>
<td>management.metrics.export.statsd.port</td>
<td>8125.0</td>
<td>Port of the StatsD server to receive exported metrics.</td>
</tr>
<tr>
<td>management.metrics.export.statsd.protocol</td>
<td>udp</td>
<td>Protocol of the StatsD server to receive exported metrics.</td>
</tr>
<tr>
<td>management.metrics.export.statsd.publish-unchanged-meters</td>
<td>true</td>
<td>Whether to send unchanged meters to the StatsD server.</td>
</tr>
<tr>
<td>management.metrics.export.wavefront.api-token</td>
<td></td>
<td>API token used when publishing metrics directly to the Wavefront API host.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.export.wavefront.batch-size</td>
<td>10000.0</td>
<td>Number of measurements per request to use for this backend. If more measurements are found, then multiple requests will be made.</td>
</tr>
<tr>
<td>management.metrics.export.wavefront.enabled</td>
<td>true</td>
<td>Whether exporting of metrics to this backend is enabled.</td>
</tr>
<tr>
<td>management.metrics.export.wavefront.global-prefix</td>
<td></td>
<td>Global prefix to separate metrics originating from this app’s white box instrumentation from those originating from other Wavefront integrations when viewed in the Wavefront UI.</td>
</tr>
<tr>
<td>management.metrics.export.wavefront.sender.flush-interval</td>
<td>1s</td>
<td></td>
</tr>
<tr>
<td>management.metrics.export.wavefront.sender.max-queue-size</td>
<td>50000.0</td>
<td></td>
</tr>
<tr>
<td>management.metrics.export.wavefront.sender.message-size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>management.metrics.export.wavefront.source</td>
<td></td>
<td>Unique identifier for the app instance that is the source of metrics being published to Wavefront. Defaults to the local host name.</td>
</tr>
<tr>
<td>management.metrics.export.wavefront.step</td>
<td>1m</td>
<td>Step size (i.e. reporting frequency) to use.</td>
</tr>
<tr>
<td>management.metrics.export.wavefront.uri</td>
<td><a href="https://longboard.wavefront.com">https://longboard.wavefront.com</a></td>
<td>URI to ship metrics to.</td>
</tr>
<tr>
<td>management.metrics.tags.*</td>
<td></td>
<td>Common tags that are applied to every meter.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>management.metrics.use-global-registry</code></td>
<td>true</td>
<td>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.</td>
</tr>
<tr>
<td><code>management.metrics.web.client.max-uri-tags</code></td>
<td>100.0</td>
<td>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.</td>
</tr>
<tr>
<td><code>management.metrics.web.client.request.autotime.enabled</code></td>
<td>true</td>
<td>Whether to automatically time web client requests.</td>
</tr>
<tr>
<td><code>management.metrics.web.client.request.autotime.percentiles</code></td>
<td></td>
<td>Computed non-aggregable percentiles to publish.</td>
</tr>
<tr>
<td><code>management.metrics.web.client.request.autotime.percentiles-histogram</code></td>
<td>false</td>
<td>Whether percentile histograms should be published.</td>
</tr>
<tr>
<td><code>management.metrics.web.client.request.metric-name</code></td>
<td>http.client.requests</td>
<td>Name of the metric for sent requests.</td>
</tr>
<tr>
<td><code>management.metrics.web.server.max-uri-tags</code></td>
<td>100.0</td>
<td>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.</td>
</tr>
<tr>
<td><code>management.metrics.web.server.request.autotime.enabled</code></td>
<td>true</td>
<td>Whether to automatically time web server requests.</td>
</tr>
<tr>
<td><code>management.metrics.web.server.request.autotime.percentiles</code></td>
<td></td>
<td>Computed non-aggregable percentiles to publish.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>management.metrics.web.server.request.autotime</td>
<td>false</td>
<td>Whether percentile histograms should be published.</td>
</tr>
<tr>
<td>management.metrics.web.server.request.ignore-trailing-slash</td>
<td>true</td>
<td>Whether the trailing slash should be ignored when recording metrics.</td>
</tr>
<tr>
<td>management.metrics.web.server.request.metric-name</td>
<td>http.server.requests</td>
<td>Name of the metric for received requests.</td>
</tr>
<tr>
<td>management.server.add-application-context-header</td>
<td>false</td>
<td>Add the &quot;X-Application-Context&quot; HTTP header in each response.</td>
</tr>
<tr>
<td>management.server.address</td>
<td></td>
<td>Network address to which the management endpoints should bind. Requires a custom management.server.port.</td>
</tr>
<tr>
<td>management.server.base-path</td>
<td></td>
<td>Management endpoint base path (for instance, <code>/management</code>). Requires a custom management.server.port.</td>
</tr>
<tr>
<td>management.server.port</td>
<td></td>
<td>Management endpoint HTTP port (uses the same port as the application by default). Configure a different port to use management-specific SSL.</td>
</tr>
<tr>
<td>management.server.ssl.ciphers</td>
<td></td>
<td>Supported SSL ciphers.</td>
</tr>
<tr>
<td>management.server.ssl.client-auth</td>
<td></td>
<td>Client authentication mode. Requires a trust store.</td>
</tr>
<tr>
<td>management.server.ssl.enabled</td>
<td>true</td>
<td>Whether to enable SSL support.</td>
</tr>
<tr>
<td>management.server.ssl.enabled-protocols</td>
<td></td>
<td>Enabled SSL protocols.</td>
</tr>
<tr>
<td>management.server.ssl.key-alias</td>
<td></td>
<td>Alias that identifies the key in the key store.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>management.server.ssl.key-password</td>
<td></td>
<td>Password used to access the key in the key store.</td>
</tr>
<tr>
<td>management.server.ssl.key-store</td>
<td></td>
<td>Path to the key store that holds the SSL certificate (typically a jks file).</td>
</tr>
<tr>
<td>management.server.ssl.key-store-password</td>
<td></td>
<td>Password used to access the key store.</td>
</tr>
<tr>
<td>management.server.ssl.key-store-provider</td>
<td></td>
<td>Provider for the key store.</td>
</tr>
<tr>
<td>management.server.ssl.key-store-type</td>
<td></td>
<td>Type of the key store.</td>
</tr>
<tr>
<td>management.server.ssl.protocol</td>
<td>TLS</td>
<td>SSL protocol to use.</td>
</tr>
<tr>
<td>management.server.ssl.trust-store</td>
<td></td>
<td>Trust store that holds SSL certificates.</td>
</tr>
<tr>
<td>management.server.ssl.trust-store-password</td>
<td></td>
<td>Password used to access the trust store.</td>
</tr>
<tr>
<td>management.server.ssl.trust-store-provider</td>
<td></td>
<td>Provider for the trust store.</td>
</tr>
<tr>
<td>management.server.ssl.trust-store-type</td>
<td></td>
<td>Type of the trust store.</td>
</tr>
<tr>
<td>management.trace.http.enabled</td>
<td>true</td>
<td>Whether to enable HTTP request-response tracing.</td>
</tr>
<tr>
<td>management.trace.http.include</td>
<td>[request-headers, response-headers, errors]</td>
<td>Items to be included in the trace. Defaults to request headers (excluding Authorization and Cookie), response headers (excluding Set-Cookie), and time taken.</td>
</tr>
</tbody>
</table>

### 10.A.15. Devtools Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.devtools.add-properties</td>
<td>true</td>
<td>Whether to enable development property defaults.</td>
</tr>
<tr>
<td>Key</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>spring.devtools.livereload.enabled</code></td>
<td>true</td>
<td>Whether to enable a livereload.com-compatible server.</td>
</tr>
<tr>
<td><code>spring.devtools.livereload.port</code></td>
<td>35729.0</td>
<td>Server port.</td>
</tr>
<tr>
<td><code>spring.devtools.remote.context-path</code></td>
<td>/~~spring-boot!~</td>
<td>Context path used to handle the remote connection.</td>
</tr>
<tr>
<td><code>spring.devtools.remote.proxy.host</code></td>
<td></td>
<td>The host of the proxy to use to connect to the remote application.</td>
</tr>
<tr>
<td><code>spring.devtools.remote.proxy.port</code></td>
<td></td>
<td>The port of the proxy to use to connect to the remote application.</td>
</tr>
<tr>
<td><code>spring.devtools.remote.restart.enabled</code></td>
<td>true</td>
<td>Whether to enable remote restart.</td>
</tr>
<tr>
<td><code>spring.devtools.remote.secret</code></td>
<td></td>
<td>A shared secret required to establish a connection (required to enable remote support).</td>
</tr>
<tr>
<td><code>spring.devtools.remote.secret-header-name</code></td>
<td>X-AUTH-TOKEN</td>
<td>HTTP header used to transfer the shared secret.</td>
</tr>
<tr>
<td><code>spring.devtools.restart.additional-exclude</code></td>
<td></td>
<td>Additional patterns that should be excluded from triggering a full restart.</td>
</tr>
<tr>
<td><code>spring.devtools.restart.additional-paths</code></td>
<td></td>
<td>Additional paths to watch for changes.</td>
</tr>
<tr>
<td><code>spring.devtools.restart.enabled</code></td>
<td>true</td>
<td>Whether to enable automatic restart.</td>
</tr>
<tr>
<td><code>spring.devtools.restart.exclude</code></td>
<td>META-INF/maven/<strong>,META-INF/resources/</strong>,resources/<strong>,static/</strong>,public/<strong>,templates/</strong>,<strong>/*.class,</strong>/*Tests.class,git.properties,META-INF/build-info.properties</td>
<td>Patterns that should be excluded from triggering a full restart.</td>
</tr>
<tr>
<td><code>spring.devtools.restart.log-condition-evaluation-delta</code></td>
<td>true</td>
<td>Whether to log the condition evaluation delta upon restart.</td>
</tr>
</tbody>
</table>
### 10.A.16. Testing Properties

<table>
<thead>
<tr>
<th>Key</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.test.database.replace</td>
<td>any</td>
<td>Type of existing DataSource to replace.</td>
</tr>
<tr>
<td>spring.test.mockmvc.print</td>
<td>default</td>
<td>MVC Print option.</td>
</tr>
</tbody>
</table>

### 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.yml` 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.

#### 10.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 database schema."
} ...
]"}
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 `application.properties`, as follows:

```
server.port=9090
server.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.

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The full name of the group. This attribute is mandatory.</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>The class name of the data type of the group. For example, if the group were based on a class annotated with <code>@ConfigurationProperties</code>, the attribute would contain the fully qualified name of that class. If it were based on a <code>@Bean</code> method, it would be the return type of that method. If the type is not known, the attribute may be omitted.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Purpose</td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>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 (.)</td>
</tr>
<tr>
<td>sourceType</td>
<td>String</td>
<td>The class name of the source that contributed this group. For example, if the group were based on a <code>@Bean</code> method annotated with <code>@ConfigurationProperties</code>, this attribute would contain the fully qualified name of the <code>@Configuration</code> class that contains the method. If the source type is not known, the attribute may be omitted.</td>
</tr>
<tr>
<td>sourceMethod</td>
<td>String</td>
<td>The full name of the method (include parenthesis and argument types) that contributed this group (for example, the name of a <code>@ConfigurationProperties</code> annotated <code>@Bean</code> method). If the source method is not known, it may be omitted.</td>
</tr>
</tbody>
</table>

### Property Attributes

The JSON object contained in the `properties` array can contain the attributes described in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The full name of the property. Names are in lower-case period-separated form (for example, <code>server.address</code>). This attribute is mandatory.</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>The full signature of the data type of the property (for example, <code>java.lang.String</code>) but also a full generic type (such as <code>java.util.Map&lt;java.lang.String,acme.MyEnum&gt;</code>). 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, <code>boolean</code> becomes <code>java.lang.Boolean</code>). Note that this class may be a complex type that gets converted from a <code>String</code> as values are bound. If the type is not known, it may be omitted.</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>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 (.)</td>
</tr>
<tr>
<td>sourceType</td>
<td>String</td>
<td>The class name of the source that contributed this property. For example, if the property were from a class annotated with <code>@ConfigurationProperties</code>, this attribute would contain the fully qualified name of that class. If the source type is unknown, it may be omitted.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Purpose</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>defaultValue</td>
<td>Object</td>
<td>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.</td>
</tr>
<tr>
<td>deprecation</td>
<td>Deprecation</td>
<td>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.</td>
</tr>
</tbody>
</table>

The JSON object contained in the deprecation attribute of each properties element can contain the following attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
<td>String</td>
<td>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.</td>
</tr>
<tr>
<td>reason</td>
<td>String</td>
<td>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 (.).</td>
</tr>
<tr>
<td>replacement</td>
<td>String</td>
<td>The full name of the property that replaces this deprecated property. If there is no replacement for this property, it may be omitted.</td>
</tr>
</tbody>
</table>

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 app.acme.target property was confusing and was renamed to app.acme.name. The following example shows how to handle that situation:
@ConfigurationProperties("app.acme")
public class AcmeProperties {
    private String name;

    public String getName() { ... }
    public void setName(String name) { ... }

    @DeprecatedConfigurationProperty(replacement = "app.acme.name")
    @Deprecated
    public String getTarget() {
        return getName();
    }

    @Deprecated
    public void setTarget(String target) {
        setName(target);
    }
}

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>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 systemcontexts), the hint either applies to the keys of the map (systemcontexts.keys) or the values (systemcontexts.values) of the map. This attribute is mandatory.</td>
</tr>
<tr>
<td>values</td>
<td>ValueHint[]</td>
<td>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.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Purpose</td>
</tr>
<tr>
<td>----------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>providers</td>
<td>ValueProvider[ ]</td>
<td>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.</td>
</tr>
</tbody>
</table>

The JSON object contained in the values attribute of each hint element can contain the attributes described in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Object</td>
<td>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.</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>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 (.)</td>
</tr>
</tbody>
</table>

The JSON object contained in the providers attribute of each hint element can contain the attributes described in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the provider to use to offer additional content assistance for the element to which the hint refers.</td>
</tr>
<tr>
<td>parameters</td>
<td>JSON object</td>
<td>Any additional parameter that the provider supports (check the documentation of the provider for more details).</td>
</tr>
</tbody>
</table>

**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.

**10.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 `sample.contexts` maps magic `String` values to an integer, as shown in the following example:

```java
@ConfigurationProperties("sample")
public class SampleProperties {
    private Map<String,Integer> contexts;
    // getters and setters
}
```

The magic values are (in this example) `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:

```json
{"hints": [
    {
        "name": "sample.contexts.keys",
        "values": [
            {
                "value": "sample1"
            },
            {
                "value": "sample2"
            }
        ]
    }
]}
```

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.
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>Permits any additional value to be provided.</td>
</tr>
<tr>
<td>class-reference</td>
<td>Auto-completes the classes available in the project. Usually constrained</td>
</tr>
<tr>
<td></td>
<td>by a base class that is specified by the target parameter.</td>
</tr>
<tr>
<td>handle-as</td>
<td>Handles the property as if it were defined by the type defined by</td>
</tr>
<tr>
<td></td>
<td>the mandatory target parameter.</td>
</tr>
<tr>
<td>logger-name</td>
<td>Auto-completes valid logger names and logger groups. Typically, package</td>
</tr>
<tr>
<td></td>
<td>and class names available in the current project can be auto-completed as</td>
</tr>
<tr>
<td></td>
<td>well as defined groups.</td>
</tr>
<tr>
<td>spring-bean-reference</td>
<td>Auto-completes the available bean names in the current project. Usually</td>
</tr>
<tr>
<td></td>
<td>constrained by a base class that is specified by the target parameter.</td>
</tr>
<tr>
<td>spring-profile-name</td>
<td>Auto-completes the available Spring profile names in the project.</td>
</tr>
</tbody>
</table>

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:
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>String (Class)</td>
<td>none</td>
<td>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.</td>
</tr>
<tr>
<td>concrete</td>
<td>boolean</td>
<td>true</td>
<td>Specify whether only concrete classes are to be considered as valid candidates.</td>
</tr>
</tbody>
</table>

The following metadata snippet corresponds to the standard server.servlet.jsp.class-name property that defines the JspServlet class name to use:
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>String (Class)</td>
<td>none</td>
<td>The fully qualified name of the type to consider for the property. This parameter is mandatory.</td>
</tr>
</tbody>
</table>

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`)

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 an `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.
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>boolean</td>
<td>true</td>
<td>Specify whether known groups should be considered.</td>
</tr>
</tbody>
</table>

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.

```json
{"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": "logger-1",
        "description": "Custom logger group for custom logging."
      }
    ]
  }
]}
```
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:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>String (Class)</td>
<td>none</td>
<td>The fully qualified name of the bean class that should be assignable to the candidate. Typically used to filter out non-candidate beans.</td>
</tr>
</tbody>
</table>

The following metadata snippet corresponds to the standard `spring.jmx.server` property that defines the name of the `MBeanServer` bean to use:

```json
{"hints": [
  {
    "name": "spring.jmx.server",
    "providers": [
      {
        "name": "spring-bean-reference",
        "parameters": {
          "target": "javax.management.MBeanServer"
        }
      }
    ]
  }
]}
```

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:

```json
{"hints": [
  {
    "name": "spring.profiles.active",
    "providers": [
      {
        "name": "spring-profile-name"
      }
    ]
  }
]}
```
10.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:

```xml
<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:

```gradle
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:

```gradle
compileJava.inputs.files(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:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <proc>none</proc>
  </configuration>
</plugin>
```

### Automatic Metadata Generation

The processor picks up both classes and methods that are annotated with `@ConfigurationProperties`. If the class is also annotated with `@ConstructorBinding`, a single constructor is expected and one property is created per constructor parameter. 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`, `@Getter`, and `@Setter` lombok annotations.

Consider the following example:
@ConfigurationProperties(prefix="server")
public class ServerProperties {

    /**
     * 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;

    // ... getter and setters
}

This exposes three properties where server.name has no default and server.ip and 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 server.ip is "IP address to listen to."

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 Enums and Collections.

For cases where the default value could not be detected, manual metadata should be provided. Consider the following example:
@ConfigurationProperties(prefix = "acme.messaging")
public class MessagingProperties {
    private List<String> addresses = new ArrayList<>(Arrays.asList("a", "b"));
    private ContainerType containerType = ContainerType.SIMPLE;
    // ... getter and setters
    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:

```json
{"properties": [
    {
        "name": "acme.messaging.addresses",
        "defaultValue": 
``"a", "b"
    },
    {
        "name": "acme.messaging.container-type",
        "defaultValue": "simple"
    }
]}
```

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:
@ConfigurationProperties(prefix="server")
public class ServerProperties {
    private String name;
    private Host host;
    // ... getter and setters
    public static class Host {
        private String ip;
        private int port;
        // ... getter and setters
    }
}

The preceding example produces metadata information for server.name, server.host.ip, and 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.

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).

### 10.C.1. spring-boot-autoconfigure

The following auto-configuration classes are from the `spring-boot-autoconfigure` module:

<table>
<thead>
<tr>
<th>Configuration Class</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveMQAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>AopAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>ApplicationAvailabilityAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>ArtemisAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>BatchAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>CacheAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>CassandraAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>CassandraDataAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>Cassandra ReactiveDataAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>Cassandra ReactiveRepositoriesAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>CassandraRepositoriesAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>ClientHttpConnectorAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>CodecsAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>ConfigurationPropertiesAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>CouchbaseAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>CouchbaseDataAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>Couchbase ReactiveDataAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>Couchbase ReactiveRepositoriesAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>CouchbaseRepositoriesAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>DataSourceAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>DataSourceTransactionManagerAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>DispatcherServletAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>ElasticsearchDataAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>ElasticsearchRepositoriesAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>ElasticsearchRestClientAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>EmbeddedLdapAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>EmbeddedMongoAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>EmbeddedWebServerFactoryCustomizerAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>ErrorMvcAutoConfiguration</td>
<td><a href="#">javadoc</a></td>
</tr>
<tr>
<td>Configuration Class</td>
<td>Links</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>ErrorWebFluxAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>FlywayAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>FreeMarkerAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>GroovyTemplateAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>GsonAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>H2ConsoleAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HazelcastAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HazelcastJpaDependencyAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HibernateJpaAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HttpEncodingAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HttpHandlerAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HttpMessageConvertersAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HypermediaAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>InfluxDbAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>IntegrationAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JacksonAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JdbcTemplateAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JerseyAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JmsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JmxAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JndiConnectionFactoryAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JndiDataSourceAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JoqAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JpaRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JsonbAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JtaAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>KafkaAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>LdapAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>LdapRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>LifecycleAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>LiquibaseAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MailSenderAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Configuration Class</td>
<td>Links</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>MailSenderValidatorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MessageSourceAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MongoAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MongoDataAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MongoReactiveAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MongoReactiveDataAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MongoReactiveRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MongoRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MultipartAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MustacheAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Neo4jAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Neo4jDataAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Neo4jReactiveDataAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Neo4jReactiveRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Neo4jRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>OAuth2ClientAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>OAuth2ResourceServerAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>PersistenceExceptionTranslationAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ProjectInfoAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>PropertyPlaceholderAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>QuartzAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>R2dbcAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>R2dbcDataAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>R2dbcRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>R2dbcTransactionManagerAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RSocketMessagingAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RSocketRequesterAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RSocketSecurityAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RSocketServerAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RSocketStrategiesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RabbitAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ReactiveElasticsearchRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ReactiveElasticsearchRestClientAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Configuration Class</td>
<td>Links</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>ReactiveOAuth2ClientAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ReactiveOAuth2ResourceServerAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ReactiveSecurityAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ReactiveUserDetailsServiceAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ReactiveWebServerFactoryAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RedisAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RedisReactiveAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RedisRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RepositoryRestMvcAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RestTemplateAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Saml2RelyingPartyAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SecurityAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SecurityFilterAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SendGridAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ServletWebServerFactoryAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SessionAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SolrAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SolrRepositoriesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SpringApplicationAdminJmxAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SpringDataWebAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>TaskExecutionAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>TaskSchedulingAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ThymeleafAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>TransactionAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>UserDetailsServiceAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ValidationAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>WebClientAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>WebFluxAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>WebMvcAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>WebServiceTemplateAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>WebServicesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>WebSocketMessagingAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>WebSocketReactiveAutoConfiguration</td>
<td>javadoc</td>
</tr>
</tbody>
</table>
The following auto-configuration classes are from the `spring-boot-actuator-autoconfigure` module:

<table>
<thead>
<tr>
<th>Configuration Class</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppOpticsMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>AtlasMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>AuditAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>AuditEventsEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>AvailabilityHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>AvailabilityProbesAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>BeansEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>CacheMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>CachesEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>CassandraHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>CassandraReactiveHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>CloudFoundryActuatorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>CompositeMeterRegistryAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ConditionsReportEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ConfigurationPropertiesReportEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ConnectionFactoryHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ConnectionPoolMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>CouchbaseHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>CouchbaseReactiveHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>DataSourceHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>DataSourcePoolMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>DatadogMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>DiskSpaceHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>DynatraceMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ElasticMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ElasticSearchReactiveHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ElasticSearchRestHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>WebSocketServletAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>XDataSourceAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Configuration Class</td>
<td>Links</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>EndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>EnvironmentEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>FlywayEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>GangliaMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>graphiteMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HazelcastHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HealthEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HeapDumpWebEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HibernateMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HttpClientMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HttpTraceAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HttpTraceEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>HumioMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>InfluxDbHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>InfluxMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>InfoContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>InfoEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>IntegrationGraphEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>IntegrationMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JerseyServerMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JettyMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JmsHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JmxEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JmxMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JolokiaEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>JvmMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>KafkaMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>KairosMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>LdapHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>LiquibaseEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Log4J2MetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>LogFileWebEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>LogTraceAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Configuration Class</td>
<td>Links</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>LogbackMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>LoggersEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MailHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ManagementContextAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ManagementWebSecurityAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MappingsEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MetricsEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MongoHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>MongoReactiveHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>Neo4jHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>NewRelicMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>PrometheusMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RabbitHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RabbitMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ReactiveCloudFoundryActuatorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ReactiveManagementContextAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ReactiveManagementWebSecurityAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RedisHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>RedisReactiveHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ScheduledTasksEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ServletManagementContextAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SessionsEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ShutdownEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SignalFxMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SimpleMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SolrHealthContributorAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>StackdriverMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>StartupEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>StatsdMetricsExportAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>SystemMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>ThreadDumpEndpointAutoConfiguration</td>
<td>javadoc</td>
</tr>
<tr>
<td>TomcatMetricsAutoConfiguration</td>
<td>javadoc</td>
</tr>
</tbody>
</table>
Appendix D: Test Auto-configuration Annotations

This appendix describes the `@…Test` auto-configuration annotations that Spring Boot provides to test slices of your application.

10.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:

<table>
<thead>
<tr>
<th>Test slice</th>
<th>Imported auto-configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>@DataCassandraTest</td>
<td>org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.cassandra.CassandraDataAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.cassandra.CassandraReactiveRepositoriesAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoryAutoConfiguration</td>
</tr>
</tbody>
</table>

---

Configuration Class

<p>| WavefrontMetricsExportAutoConfiguration | javadoc |
| WebEndpointAutoConfiguration            | javadoc |
| WebFluxMetricsAutoConfiguration         | javadoc |
| WebMvcMetricsAutoConfiguration          | javadoc |</p>
<table>
<thead>
<tr>
<th>Test slice</th>
<th>Imported auto-configuration</th>
</tr>
</thead>
</table>
| @DataJdbcTest | 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.transaction.TransactionAutoConfiguration  
org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration  |
| @DataJpaTest | 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.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  
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.transaction.TransactionAutoConfiguration  
org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration  
org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManagerAutoConfiguration  |
<table>
<thead>
<tr>
<th>Test slice</th>
<th>Imported auto-configuration</th>
</tr>
</thead>
</table>
| @DataMongoTest     | org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration  
|                    | org.springframework.boot.autoconfigure.data.mongodb.MongoDataAutoConfiguration  
|                    | org.springframework.boot.autoconfigure.data.mongodb.MongoReactiveDataAutoConfiguration  
|                    | org.springframework.boot.autoconfigure.data.mongodb.MongoRepositoriesAutoConfiguration  
|                    | org.springframework.boot.autoconfigure.data.mongodb.MongoAutoConfiguration  
|                    | org.springframework.boot.autoconfigure.data.mongodb.MongoAutoReactiveAutoConfiguration  
|                    | org.springframework.boot.autoconfigure.data.mongodb.embedded.EmbeddedMongoAutoConfiguration  
|                    | org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration                                                                                           |
| @DataNeo4jTest     | 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.Neo4jRepositoriesAutoConfiguration  
|                    | org.springframework.boot.autoconfigure.data.neo4j.Neo4jAutoConfiguration  
|                    | org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration                                                                                           |
| @DataR2dbcTest     | 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  
<p>|                    | org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration                                                                                           |</p>
<table>
<thead>
<tr>
<th>Test slice</th>
<th>Imported auto-configuration</th>
</tr>
</thead>
</table>
| @DataRedisTest | org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration  
osrg.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration  
osrg.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration |
| @JdbcTest | org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration  
osrg.springframework.boot.autoconfigure.data.flyway.FlywayAutoConfiguration  
osrg.springframework.boot.autoconfigure.data.jdbc.DataSourceAutoConfiguration  
osrg.springframework.boot.autoconfigure.data.jdbc DataSourceTransactionManagerAutoConfiguration  
osrg.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration  
osrg.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration  
osrg.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration  
osrg.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration |
| @JooqTest | org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration  
osrg.springframework.boot.autoconfigure.data.flyway.FlywayAutoConfiguration  
osrg.springframework.boot.autoconfigure.data.jdbc.DataSourceAutoConfiguration  
osrg.springframework.boot.autoconfigure.data.jdbc DataSourceTransactionManagerAutoConfiguration  
osrg.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration  
osrg.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration  
osrg.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration |
| @JsonTest | org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration  
osrg.springframework.boot.autoconfigure.gson.GsonAutoConfiguration  
osrg.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration  
osrg.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration  
osrg.springframework.boot.test.autoconfigure.json.JsonTestersAutoConfiguration |
<table>
<thead>
<tr>
<th>Test slice</th>
<th>Imported auto-configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>@RestClientTest</td>
<td>org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.jackson.JsonbAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.test.autoconfigure.web.client.MockRestServiceServerAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.test.autoconfigure.web.client.WebClientRestTemplateAutoConfiguration</td>
</tr>
<tr>
<td>Test slice</td>
<td>Imported auto-configuration</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>@WebFluxTest</td>
<td>org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.security.oauth2.client.reactive.ReactiveOAuth2ClientAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.ReactiveOAuth2ResourceServerAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration</td>
</tr>
<tr>
<td></td>
<td>org.springframework.boot.test.autoconfigure.web.reactive.WebTestClientAutoConfiguration</td>
</tr>
<tr>
<td>Test slice</td>
<td>Imported auto-configuration</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------</td>
</tr>
</tbody>
</table>
| `@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.hateoas.HypermediaAutoConfiguration
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.servlet.MockMvcAutoConfiguration
org.springframework.boot.test.autoconfigure.web.servlet.MockMvcSecurityConfiguration
org.springframework.boot.test.autoconfigure.web.servlet.MockMvcWebDriverAutoConfiguration
``` |
<table>
<thead>
<tr>
<th>Test slice</th>
<th>Imported auto-configuration</th>
</tr>
</thead>
</table>

**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.

### 10.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:
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:

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. It provides a list of jar names (not including the directory) in the order that they should be added to the classpath. 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:

```
- "dependency2.jar"
- "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 ("··-·") and 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/"

### 10.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 ||
+-----------------+||+-----------+----------+|
+-------------------+-------------------------+
```

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`.

### 10.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
```

You need not specify Class-Path entries in your manifest file. The classpath is deduced from the nested jars.

10.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:

<table>
<thead>
<tr>
<th>Key</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>loader.path</td>
<td>Comma-separated Classpath, such as lib,${HOME}/app/lib. Earlier entries take precedence, like a regular -classpath on the javac command line.</td>
</tr>
<tr>
<td>Key</td>
<td>Purpose</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>loader.home</td>
<td>Used to resolve relative paths in <code>loader.path</code>. For example, given <code>loader.path=lib</code>, then</td>
</tr>
<tr>
<td></td>
<td><code>${loader.home}/lib</code> is a classpath location (along with all jar files in that directory). This</td>
</tr>
<tr>
<td></td>
<td>property is also used to locate a <code>loader.properties</code> file, as in the following example <code>/opt/app</code></td>
</tr>
<tr>
<td></td>
<td>It defaults to <code>${user.dir}</code>.</td>
</tr>
<tr>
<td>loader.args</td>
<td>Default arguments for the main method (space separated).</td>
</tr>
<tr>
<td>loader.main</td>
<td>Name of main class to launch (for example, <code>com.app.Application</code>).</td>
</tr>
<tr>
<td>loader.config.name</td>
<td>Name of properties file (for example, <code>launcher</code>). It defaults to <code>loader</code>.</td>
</tr>
<tr>
<td>loader.config.location</td>
<td>Path to properties file (for example, <code>classpath:loader.properties</code>). It defaults to <code>loader.properties</code>.</td>
</tr>
<tr>
<td>loader.system</td>
<td>Boolean flag to indicate that all properties should be added to System properties. It defaults to <code>false</code>.</td>
</tr>
</tbody>
</table>

When specified as environment variables or manifest entries, the following names should be used:

<table>
<thead>
<tr>
<th>Key</th>
<th>Manifest entry</th>
<th>Environment variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>loader.path</td>
<td>Loader-Path</td>
<td>${LOADER_PATH}</td>
</tr>
<tr>
<td>loader.home</td>
<td>Loader-Home</td>
<td>${LOADER_HOME}</td>
</tr>
<tr>
<td>loader.args</td>
<td>Loader-Args</td>
<td>${LOADER_ARGS}</td>
</tr>
<tr>
<td>loader.main</td>
<td>Start-Class</td>
<td>${LOADER_MAIN}</td>
</tr>
<tr>
<td>loader.config.location</td>
<td>Loader-Config-Location</td>
<td>${LOADER_CONFIG_LOCATION}</td>
</tr>
<tr>
<td>loader.system</td>
<td>Loader-System</td>
<td>${LOADER_SYSTEM}</td>
</tr>
</tbody>
</table>

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.

### 10.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.

### 10.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.

#### 10.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.

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Artifact ID</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>antlr</td>
<td>2.7.7</td>
</tr>
<tr>
<td>ch.qos.logback</td>
<td>logback-access</td>
<td>1.2.3</td>
</tr>
<tr>
<td>ch.qos.logback</td>
<td>logback-classic</td>
<td>1.2.3</td>
</tr>
<tr>
<td>ch.qos.logback</td>
<td>logback-core</td>
<td>1.2.3</td>
</tr>
<tr>
<td>com.atomikos</td>
<td>transactions-jdbc</td>
<td>4.0.6</td>
</tr>
<tr>
<td>com.atomikos</td>
<td>transactions-jms</td>
<td>4.0.6</td>
</tr>
<tr>
<td>com.atomikos</td>
<td>transactions-jta</td>
<td>4.0.6</td>
</tr>
<tr>
<td>com.couchbase.client</td>
<td>java-client</td>
<td>3.0.10</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-core</td>
<td>4.9.0</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-core-shaded</td>
<td>4.9.0</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-mapper-processor</td>
<td>4.9.0</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-mapper-runtime</td>
<td>4.9.0</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-metrics-micrometer</td>
<td>4.9.0</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-metrics-micrometer</td>
<td>4.9.0</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-query-builder</td>
<td>4.9.0</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-query-builder</td>
<td>4.9.0</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-shaded-guava</td>
<td>25.1-jre-graal-sub-1</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>java-driver-test-infra</td>
<td>4.9.0</td>
</tr>
<tr>
<td>com.datastax.oss</td>
<td>native-protocol</td>
<td>1.4.11</td>
</tr>
<tr>
<td>com.fasterxml.classmate</td>
<td>classmate</td>
<td>1.5.1</td>
</tr>
<tr>
<td>com.fasterxml.jackson.core</td>
<td>jackson-annotations</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.core</td>
<td>jackson-core</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.core</td>
<td>jackson-databind</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.dataformat</td>
<td>jackson-dataformat-avro</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.dataformat</td>
<td>jackson-dataformat-cbor</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.dataformat</td>
<td>jackson-dataformat-csv</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.dataformat</td>
<td>jackson-dataformat- Ion</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.dataformat</td>
<td>jackson-dataformat-properties</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.dataformat</td>
<td>jackson-dataformat-protobuf</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.dataformat</td>
<td>jackson-dataformat-smile</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.dataformat</td>
<td>jackson-dataformat-xml</td>
<td>2.11.4</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>com.fasterxml.jackson.dataformat</td>
<td>jackson-dataformat-yaml</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-eclipse-collections</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-guava</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-hibernate3</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-hibernate4</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-hibernate5</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-hppc</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-jaxrs</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-jdk8</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-joda</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-joda-money</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-json-org</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-jsr310</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-jsr353</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.datatype</td>
<td>jackson-datatype-pcollections</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jaxrs</td>
<td>jackson-jaxrs-base</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jaxrs</td>
<td>jackson-jaxrs-cbor-provider</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jaxrs</td>
<td>jackson-jaxrs-json-provider</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jaxrs</td>
<td>jackson-jaxrs-smile-provider</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jaxrs</td>
<td>jackson-jaxrs-xml-provider</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jaxrs</td>
<td>jackson-jaxrs-yaml-provider</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jr</td>
<td>jackson-jr-all</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jr</td>
<td>jackson-jr-annotation-support</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jr</td>
<td>jackson-jr-objects</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jr</td>
<td>jackson-jr-retrofit2</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.jr</td>
<td>jackson-jr-stree</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-afterburner</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-guice</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-jaxb-annotations</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-jsonSchema</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-kotlin</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-mrbean</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-osgi</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-parameter-names</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-paranamer</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module</td>
<td>jackson-module-scala_2.10</td>
<td>2.11.4</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>--------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module.jakson</td>
<td>jackson-module-scala_2.11</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module.jakson</td>
<td>jackson-module-scala_2.12</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.fasterxml.jackson.module.jakson</td>
<td>jackson-module-scala_2.13</td>
<td>2.11.4</td>
</tr>
<tr>
<td>com.github.ben-manes.caffeine</td>
<td>caffeine</td>
<td>2.8.8</td>
</tr>
<tr>
<td>com.github.ben-manes.caffeine</td>
<td>guava</td>
<td>2.8.8</td>
</tr>
<tr>
<td>com.github.ben-manes.caffeine</td>
<td>jcache</td>
<td>2.8.8</td>
</tr>
<tr>
<td>com.github.ben-manes.caffeine</td>
<td>simulator</td>
<td>2.8.8</td>
</tr>
<tr>
<td>com.github.mxab.thymeleaf.extras</td>
<td>thymeleaf-extras-data-attribute</td>
<td>2.0.1</td>
</tr>
<tr>
<td>com.google.appengine</td>
<td>appengine-api-1.0-sdk</td>
<td>1.9.84</td>
</tr>
<tr>
<td>com.google.cloud</td>
<td>cloud-spanner-r2dbc</td>
<td>0.2.0</td>
</tr>
<tr>
<td>com.google.code.gson</td>
<td>gson</td>
<td>2.8.6</td>
</tr>
<tr>
<td>com.h2database</td>
<td>h2</td>
<td>1.4.200</td>
</tr>
<tr>
<td>com.hazelcast</td>
<td>hazelcast</td>
<td>4.0.3</td>
</tr>
<tr>
<td>com.hazelcast</td>
<td>hazelcast-hibernate52</td>
<td>2.1.1</td>
</tr>
<tr>
<td>com.hazelcast</td>
<td>hazelcast-hibernate53</td>
<td>2.1.1</td>
</tr>
<tr>
<td>com.hazelcast</td>
<td>hazelcast-spring</td>
<td>4.0.3</td>
</tr>
<tr>
<td>com.ibm.db2</td>
<td>jcc</td>
<td>11.5.5.0</td>
</tr>
<tr>
<td>com.jayway.jsonpath</td>
<td>json-path</td>
<td>2.4.0</td>
</tr>
<tr>
<td>com.jayway.jsonpath</td>
<td>json-path-assert</td>
<td>2.4.0</td>
</tr>
<tr>
<td>com.microsoft.sqlserver</td>
<td>mssql-jdbc</td>
<td>8.4.1.jre8</td>
</tr>
<tr>
<td>com.nimbusds</td>
<td>nimbus-jose-jwt</td>
<td>9.1.3</td>
</tr>
<tr>
<td>com.nimbusds</td>
<td>oauth2-oidc-sdk</td>
<td>8.23.1</td>
</tr>
<tr>
<td>com.oracle.jdbc.dbdebug</td>
<td>ojdbc10_g</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.dbdebug</td>
<td>ojdbc10dms_g</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.dbdebug</td>
<td>ojdbc8_g</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.dbdebug</td>
<td>ojdbc8dms_g</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.db.ha</td>
<td>ons</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.db.ha</td>
<td>simplefan</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.dbjdbc</td>
<td>ojdbc10</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.dbjdbc</td>
<td>ojdbc10-production</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.dbjdbc</td>
<td>ojdbc8</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.dbjdbc</td>
<td>ojdbc8-production</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.dbjdbc</td>
<td>ucp</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.db.jdbc.debug</td>
<td>ojdbc10-debug</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.db.jdbc.debug</td>
<td>ojdbc10-observability-debug</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.db.jdbc.debug</td>
<td>ojdbc8-debug</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.db.jdbc.debug</td>
<td>ojdbc8-observability-debug</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.jdbc.db.db.nls</td>
<td>orai18n</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>com.oracle.database.observability</td>
<td>dms</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.database.observability</td>
<td>ojdbc10-observability</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.database.observability</td>
<td>ojdbc10dms</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.database.observability</td>
<td>ojdbc8-observability</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.database.observability</td>
<td>ojdbc8dms</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.database.security</td>
<td>oraclepki</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.database.security</td>
<td>osdt_cert</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.database.security</td>
<td>osdt_core</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.database.xml</td>
<td>xdb</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.database.xml</td>
<td>xmlparserv2</td>
<td>19.8.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>dms</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ojdbc10</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ojdbc10_g</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ojdbc10dms</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ojdbc10dms_g</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ojdbc8</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ojdbc8_g</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ojdbc8dms</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ojdbc8dms_g</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ons</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>oraclepki</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>orai18n</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>osdt_cert</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>osdt_core</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>simplefan</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>ucp</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>xdb</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.oracle.ojdbc</td>
<td>xmlparserv2</td>
<td>19.3.0.0</td>
</tr>
<tr>
<td>com.querydsl</td>
<td>querydsl-apt</td>
<td>4.4.0</td>
</tr>
<tr>
<td>com.querydsl</td>
<td>querydsl-collections</td>
<td>4.4.0</td>
</tr>
<tr>
<td>com.querydsl</td>
<td>querydsl-core</td>
<td>4.4.0</td>
</tr>
<tr>
<td>com.querydsl</td>
<td>querydsl-jpa</td>
<td>4.4.0</td>
</tr>
<tr>
<td>com.querydsl</td>
<td>querydsl-mongodb</td>
<td>4.4.0</td>
</tr>
<tr>
<td>com.rabbitmq</td>
<td>amqp-client</td>
<td>5.10.0</td>
</tr>
<tr>
<td>com.samskivert</td>
<td>jmustache</td>
<td>1.15</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>com.sendgrid</td>
<td>sendgrid-java</td>
<td>4.6.8</td>
</tr>
<tr>
<td>com.squareup.okhttp3</td>
<td>logging-interceptor</td>
<td>3.14.9</td>
</tr>
<tr>
<td>com.squareup.okhttp3</td>
<td>mockwebserver</td>
<td>3.14.9</td>
</tr>
<tr>
<td>com.squareup.okhttp3</td>
<td>okcurl</td>
<td>3.14.9</td>
</tr>
<tr>
<td>com.squareup.okhttp3</td>
<td>okhttp</td>
<td>3.14.9</td>
</tr>
<tr>
<td>com.squareup.okhttp3</td>
<td>okhttp-dnsoverhttps</td>
<td>3.14.9</td>
</tr>
<tr>
<td>com.squareup.okhttp3</td>
<td>okhttp-sse</td>
<td>3.14.9</td>
</tr>
<tr>
<td>com.squareup.okhttp3</td>
<td>okhttp-testing-support</td>
<td>3.14.9</td>
</tr>
<tr>
<td>com.squareup.okhttp3</td>
<td>okhttp-tls</td>
<td>3.14.9</td>
</tr>
<tr>
<td>com.sun.activation</td>
<td>jakarta.activation</td>
<td>1.2.2</td>
</tr>
<tr>
<td>com.sun.mail</td>
<td>jakarta.mail</td>
<td>1.6.5</td>
</tr>
<tr>
<td>com.sun.xml.messaging.saaj</td>
<td>saaj-impl</td>
<td>1.5.2</td>
</tr>
<tr>
<td>com.unboundid</td>
<td>unboundid-ldapsdk</td>
<td>4.0.14</td>
</tr>
<tr>
<td>com.zaxxer</td>
<td>HikariCP</td>
<td>3.4.5</td>
</tr>
<tr>
<td>commons-codec</td>
<td>commons-codec</td>
<td>1.15</td>
</tr>
<tr>
<td>commons-pool</td>
<td>commons-pool</td>
<td>1.6</td>
</tr>
<tr>
<td>de.flapdoodle.embed</td>
<td>de.flapdoodle.embed.mongo</td>
<td>2.2.0</td>
</tr>
<tr>
<td>dev.miku</td>
<td>r2dbc-mysql</td>
<td>0.8.2.RELEASE</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-annotation</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-caffeine</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-collectd</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-core</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-ehcache</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-graphite</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-healthchecks</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-httpasyncclient</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-httpclient</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-jcache</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-jdbi</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-jdbi3</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-jersey2</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-jetty9</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-jmx</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-jvm</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-log4j2</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-logback</td>
<td>4.1.17</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-servlet</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.dropwizard.metrics</td>
<td>metrics-servlets</td>
<td>4.1.17</td>
</tr>
<tr>
<td>io.lettuce</td>
<td>lettuce-core</td>
<td>6.0.2.RELEASE</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-core</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-jersey2</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-appoptics</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-atlas</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-azuremonitor</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-cloudwatch</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-cloudwatch2</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-datadog</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-dynatrace</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-elastic</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-ganglia</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-graphite</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-health</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-humio</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-influx</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-jmx</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-kairos</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-new-relic</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-opentsdb</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-prometheus</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-signalfx</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-stackdriver</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-statsd</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-registry-wavefront</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.micrometer</td>
<td>micrometer-test</td>
<td>1.6.3</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-all</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-buffer</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-dns</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-haproxy</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-http</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-http2</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-memcache</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-mqtt</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-redis</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-smtp</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-socks</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-stomp</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-codec-xml</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-common</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-dev-tools</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-example</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-handler</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-handler-proxy</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-resolver</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-resolver-dns</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-resolver-dns-native-macos</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-tcnative</td>
<td>2.0.36.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-tcnative-boringssl-static</td>
<td>2.0.36.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-transport</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-transport-native-epoll</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-transport-native-kqueue</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-transport-native-unix-common</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-transport-rxtx</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-transport-sctp</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.netty</td>
<td>netty-transport-udt</td>
<td>4.1.58.Final</td>
</tr>
<tr>
<td>io.projectreactor</td>
<td>reactor-core</td>
<td>3.4.2</td>
</tr>
<tr>
<td>io.projectreactor</td>
<td>reactor-test</td>
<td>3.4.2</td>
</tr>
<tr>
<td>io.projectreactor</td>
<td>reactor-tools</td>
<td>3.4.2</td>
</tr>
<tr>
<td>io.projectreactor.addons</td>
<td>reactor-adapter</td>
<td>3.4.2</td>
</tr>
<tr>
<td>io.projectreactor.addons</td>
<td>reactor-extra</td>
<td>3.4.2</td>
</tr>
<tr>
<td>io.projectreactor.addons</td>
<td>reactor-pool</td>
<td>0.2.2</td>
</tr>
<tr>
<td>io.projectreactor.kafka</td>
<td>reactor-kafka</td>
<td>1.3.1</td>
</tr>
<tr>
<td>io.projectreactor.kotlin</td>
<td>reactor-kotlin-extensions</td>
<td>1.1.2</td>
</tr>
<tr>
<td>io.projectreactor.netty</td>
<td>reactor-netty</td>
<td>1.0.3</td>
</tr>
<tr>
<td>io.projectreactor.netty</td>
<td>reactor-netty-core</td>
<td>1.0.3</td>
</tr>
<tr>
<td>io.projectreactor.netty</td>
<td>reactor-netty-http</td>
<td>1.0.3</td>
</tr>
<tr>
<td>io.projectreactor.netty</td>
<td>reactor-netty-http-brave</td>
<td>1.0.3</td>
</tr>
<tr>
<td>io.projectreactor.rabbitmq</td>
<td>reactor-rabbitmq</td>
<td>1.5.1</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>io.prometheus</td>
<td>simpleclient_pushgateway</td>
<td>0.9.0</td>
</tr>
<tr>
<td>io.r2dbc</td>
<td>r2dbc-h2</td>
<td>0.8.4.RELEASE</td>
</tr>
<tr>
<td>io.r2dbc</td>
<td>r2dbc-mssql</td>
<td>0.8.5.RELEASE</td>
</tr>
<tr>
<td>io.r2dbc</td>
<td>r2dbc-pool</td>
<td>0.8.5.RELEASE</td>
</tr>
<tr>
<td>io.r2dbc</td>
<td>r2dbc-postgresql</td>
<td>0.8.6.RELEASE</td>
</tr>
<tr>
<td>io.r2dbc</td>
<td>r2dbc-proxy</td>
<td>0.8.3.RELEASE</td>
</tr>
<tr>
<td>io.r2dbc</td>
<td>r2dbc-spi</td>
<td>0.8.3.RELEASE</td>
</tr>
<tr>
<td>io.reactivex</td>
<td>rxjava</td>
<td>1.3.8</td>
</tr>
<tr>
<td>io.reactivex</td>
<td>rxjava-reactive-streams</td>
<td>1.2.1</td>
</tr>
<tr>
<td>io.reactivex</td>
<td>rxjava2</td>
<td>2.2.20</td>
</tr>
<tr>
<td>io.rest-assured</td>
<td>json-path</td>
<td>3.3.0</td>
</tr>
<tr>
<td>io.rest-assured</td>
<td>json-schema-validator</td>
<td>3.3.0</td>
</tr>
<tr>
<td>io.rest-assured</td>
<td>rest-assured</td>
<td>3.3.0</td>
</tr>
<tr>
<td>io.rest-assured</td>
<td>scala-support</td>
<td>3.3.0</td>
</tr>
<tr>
<td>io.rest-assured</td>
<td>spring-mock-mvc</td>
<td>3.3.0</td>
</tr>
<tr>
<td>io.rest-assured</td>
<td>spring-web-test-client</td>
<td>3.3.0</td>
</tr>
<tr>
<td>io.rest-assured</td>
<td>xml-path</td>
<td>3.3.0</td>
</tr>
<tr>
<td>io.rsocket</td>
<td>rsocket-core</td>
<td>1.1.0</td>
</tr>
<tr>
<td>io.rsocket</td>
<td>rsocket-load-balancer</td>
<td>1.1.0</td>
</tr>
<tr>
<td>io.rsocket</td>
<td>rsocket-micrometer</td>
<td>1.1.0</td>
</tr>
<tr>
<td>io.rsocket</td>
<td>rsocket-test</td>
<td>1.1.0</td>
</tr>
<tr>
<td>io.rsocket</td>
<td>rsocket-transport-local</td>
<td>1.1.0</td>
</tr>
<tr>
<td>io.rsocket</td>
<td>rsocket-transport-netty</td>
<td>1.1.0</td>
</tr>
<tr>
<td>io.spring.gradle</td>
<td>dependency-management-plugin</td>
<td>1.0.11.RELEASE</td>
</tr>
<tr>
<td>io.undertow</td>
<td>undertow-core</td>
<td>2.2.3.Final</td>
</tr>
<tr>
<td>io.undertow</td>
<td>undertow-servlet</td>
<td>2.2.3.Final</td>
</tr>
<tr>
<td>io.undertow</td>
<td>undertow-websockets-jsr</td>
<td>2.2.3.Final</td>
</tr>
<tr>
<td>jakarta.activation</td>
<td>jakarta.activation-api</td>
<td>1.2.2</td>
</tr>
<tr>
<td>jakarta.annotation</td>
<td>jakarta.annotation-api</td>
<td>1.3.5</td>
</tr>
<tr>
<td>jakarta.jms</td>
<td>jakarta.jms-api</td>
<td>2.0.3</td>
</tr>
<tr>
<td>jakarta.json</td>
<td>jakarta.json-api</td>
<td>1.1.6</td>
</tr>
<tr>
<td>jakarta.json.bind</td>
<td>jakarta.json.bind-api</td>
<td>1.0.2</td>
</tr>
<tr>
<td>jakarta.mail</td>
<td>jakarta.mail-api</td>
<td>1.6.5</td>
</tr>
<tr>
<td>jakarta.persistence</td>
<td>jakarta.persistence-api</td>
<td>2.2.3</td>
</tr>
<tr>
<td>jakarta.servlet</td>
<td>jakarta.servlet-api</td>
<td>4.0.4</td>
</tr>
<tr>
<td>jakarta.servlet.jsp.jstl</td>
<td>jakarta.servlet.jsp.jstl-api</td>
<td>1.2.7</td>
</tr>
<tr>
<td>jakarta.transaction</td>
<td>jakarta.transaction-api</td>
<td>1.3.3</td>
</tr>
<tr>
<td>jakarta.validation</td>
<td>jakarta.validation-api</td>
<td>2.0.2</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>jakarta.websocket</td>
<td>jakarta.websocket-api</td>
<td>1.1.2</td>
</tr>
<tr>
<td>jakarta.ws.rs</td>
<td>jakarta.ws.rs-api</td>
<td>2.1.6</td>
</tr>
<tr>
<td>jakarta.xml.bind</td>
<td>jakarta.xml.bind-api</td>
<td>2.3.3</td>
</tr>
<tr>
<td>jakarta.xml.soap</td>
<td>jakarta.xml.soap-api</td>
<td>1.4.2</td>
</tr>
<tr>
<td>jakarta.xml.ws</td>
<td>jakarta.xml.ws-api</td>
<td>2.3.3</td>
</tr>
<tr>
<td>javax.activation</td>
<td>javax.activation-api</td>
<td>1.2.0</td>
</tr>
<tr>
<td>javax.annotation</td>
<td>javax.annotation-api</td>
<td>1.3.2</td>
</tr>
<tr>
<td>javax.cache</td>
<td>cache-api</td>
<td>1.1.1</td>
</tr>
<tr>
<td>javax.jms</td>
<td>javax.jms-api</td>
<td>2.0.1</td>
</tr>
<tr>
<td>javax.json</td>
<td>javax.json-api</td>
<td>1.1.4</td>
</tr>
<tr>
<td>javax.json.bind</td>
<td>javax.json.bind-api</td>
<td>1.0</td>
</tr>
<tr>
<td>javax.mail</td>
<td>javax.mail-api</td>
<td>1.6.2</td>
</tr>
<tr>
<td>javax.money</td>
<td>money-api</td>
<td>1.1</td>
</tr>
<tr>
<td>javax.persistence</td>
<td>javax.persistence-api</td>
<td>2.2</td>
</tr>
<tr>
<td>javax.servlet</td>
<td>javax.servlet-api</td>
<td>4.0.1</td>
</tr>
<tr>
<td>javax.servlet</td>
<td>jstl</td>
<td>1.2</td>
</tr>
<tr>
<td>javax.transaction</td>
<td>javax.transaction-api</td>
<td>1.3</td>
</tr>
<tr>
<td>javax.validation</td>
<td>validation-api</td>
<td>2.0.1.Final</td>
</tr>
<tr>
<td>javax.websocket</td>
<td>javax.websocket-api</td>
<td>1.1</td>
</tr>
<tr>
<td>javax.xml.bind</td>
<td>jaxb-api</td>
<td>2.3.1</td>
</tr>
<tr>
<td>javax.xml.ws</td>
<td>jaxws-api</td>
<td>2.3.1</td>
</tr>
<tr>
<td>jaxen</td>
<td>jaxen</td>
<td>1.2.0</td>
</tr>
<tr>
<td>junit</td>
<td>junit</td>
<td>4.12</td>
</tr>
<tr>
<td>mysql</td>
<td>mysql-connector-java</td>
<td>8.0.22</td>
</tr>
<tr>
<td>net.bytebuddy</td>
<td>byte-buddy</td>
<td>1.10.19</td>
</tr>
<tr>
<td>net.bytebuddy</td>
<td>byte-buddy-agent</td>
<td>1.10.19</td>
</tr>
<tr>
<td>net.minidev</td>
<td>json-smart</td>
<td>2.3</td>
</tr>
<tr>
<td>net.sf.ehcache</td>
<td>ehcache</td>
<td>2.10.6</td>
</tr>
<tr>
<td>net.sourceforge.htmlunit</td>
<td>htmlunit</td>
<td>2.44.0</td>
</tr>
<tr>
<td>net.sourceforge.jtds</td>
<td>jtds</td>
<td>1.3.1</td>
</tr>
<tr>
<td>net.sourceforge.nekohtml</td>
<td>nekohtml</td>
<td>1.9.22</td>
</tr>
<tr>
<td>nz.net.ultraq.thymeleaf</td>
<td>thymeleaf-layout-dialect</td>
<td>2.5.2</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-amqp</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-blueprint</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-broker</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-camel</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-client</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-console</td>
<td>5.16.0</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-http</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-jas</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-jdbc-store</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-jms-pool</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-kahadb-store</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-karaf</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-leveldb-store</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-log4j-appender</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-mqtt</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-openwire-generator</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-openwire-legacy</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-osgi</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-partition</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-pool</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-ra</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-run</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-runtime-config</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq.shiro</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-spring</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-stomp</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>activemq-web</td>
<td>5.16.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>artemis-amqp-protocol</td>
<td>2.15.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>artemis-commons</td>
<td>2.15.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>artemis-core-client</td>
<td>2.15.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>artemis-jms-client</td>
<td>2.15.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>artemis-jms-server</td>
<td>2.15.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>artemis-journal</td>
<td>2.15.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>artemis-selector</td>
<td>2.15.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>artemis-server</td>
<td>2.15.0</td>
</tr>
<tr>
<td>org.apache.activemq</td>
<td>artemis-service-extensions</td>
<td>2.15.0</td>
</tr>
<tr>
<td>org.apache.commons</td>
<td>commons-dbcp2</td>
<td>2.8.0</td>
</tr>
<tr>
<td>org.apache.commons</td>
<td>commons-lang3</td>
<td>3.11</td>
</tr>
<tr>
<td>org.apache.commons</td>
<td>commons-pool2</td>
<td>2.9.0</td>
</tr>
<tr>
<td>org.apache.derby</td>
<td>derby</td>
<td>10.14.2.0</td>
</tr>
<tr>
<td>org.apache.derby</td>
<td>derbyclient</td>
<td>10.14.2.0</td>
</tr>
<tr>
<td>org.apache.httpcomponents</td>
<td>fluent-hc</td>
<td>4.5.13</td>
</tr>
<tr>
<td>org.apache.httpcomponents</td>
<td>httpasyncclient</td>
<td>4.1.4</td>
</tr>
<tr>
<td>org.apache.httpcomponents</td>
<td>httpclient</td>
<td>4.5.13</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.apache.httpcomponents</td>
<td>httpclient-cache</td>
<td>4.5.13</td>
</tr>
<tr>
<td>org.apache.httpcomponents</td>
<td>httpclient-osgi</td>
<td>4.5.13</td>
</tr>
<tr>
<td>org.apache.httpcomponents</td>
<td>httpclient-win</td>
<td>4.5.13</td>
</tr>
<tr>
<td>org.apache.httpcomponents</td>
<td>httpcore</td>
<td>4.4.14</td>
</tr>
<tr>
<td>org.apache.httpcomponents</td>
<td>httpcore-nio</td>
<td>4.4.14</td>
</tr>
<tr>
<td>org.apache.httpcomponents</td>
<td>httpmime</td>
<td>4.5.13</td>
</tr>
<tr>
<td>org.apache.johnzon</td>
<td>johnzon-core</td>
<td>1.2.9</td>
</tr>
<tr>
<td>org.apache.johnzon</td>
<td>johnzon-jaxrs</td>
<td>1.2.9</td>
</tr>
<tr>
<td>org.apache.johnzon</td>
<td>johnzon-jsonb</td>
<td>1.2.9</td>
</tr>
<tr>
<td>org.apache.johnzon</td>
<td>johnzon-jsonb-extras</td>
<td>1.2.9</td>
</tr>
<tr>
<td>org.apache.johnzon</td>
<td>johnzon-jsonschema</td>
<td>1.2.9</td>
</tr>
<tr>
<td>org.apache.johnzon</td>
<td>johnzon-mapper</td>
<td>1.2.9</td>
</tr>
<tr>
<td>org.apache.johnzon</td>
<td>johnzon-websocket</td>
<td>1.2.9</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>connect-api</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>connect-basic-auth-extension</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>connect-file</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>connect-json</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>connect-runtime</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>connect-transforms</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>kafka-clients</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>kafka-log4j-appender</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>kafka-streams</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>kafka-streams-scala_2.12</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>kafka-streams-scala_2.13</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>kafka-streams-test-utils</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>kafka-tools</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>kafka_2.12</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.kafka</td>
<td>kafka_2.13</td>
<td>2.6.0</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-1.2-api</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-api</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-appserver</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-cassandra</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-core</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-couchdb</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-docker</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-flume-ng</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-iostreams</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-jcl</td>
<td>2.13.3</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-jmx-gui</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-jpa</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-jpl</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-jul</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-kubernetes</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-liquibase</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-mongodb2</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-mongodb3</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-slf4j-impl</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-slf4j18-impl</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-spring-cloud-config-client</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-taglib</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-to-slf4j</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.logging.log4j</td>
<td>log4j-web</td>
<td>2.13.3</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-analysis-extras</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-analytics</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-cell</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-clustering</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-core</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-dataimporthandler</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-dataimporthandler-extras</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-langid</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-ltr</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-solrj</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-test-framework</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.solr</td>
<td>solr-velocity</td>
<td>8.5.2</td>
</tr>
<tr>
<td>org.apache.tomcat</td>
<td>tomcat-annotations-api</td>
<td>9.0.41</td>
</tr>
<tr>
<td>org.apache.tomcat</td>
<td>tomcat-jdbc</td>
<td>9.0.41</td>
</tr>
<tr>
<td>org.apache.tomcat</td>
<td>tomcat-jsp-api</td>
<td>9.0.41</td>
</tr>
<tr>
<td>org.apache.tomcat.embed</td>
<td>tomcat-embed-core</td>
<td>9.0.41</td>
</tr>
<tr>
<td>org.apache.tomcat.embed</td>
<td>tomcat-embed-el</td>
<td>9.0.41</td>
</tr>
<tr>
<td>org.apache.tomcat.embed</td>
<td>tomcat-embed-jasper</td>
<td>9.0.41</td>
</tr>
<tr>
<td>org.apache.tomcat.embed</td>
<td>tomcat-embed-websocket</td>
<td>9.0.41</td>
</tr>
<tr>
<td>org.aspectj</td>
<td>aspectjrt</td>
<td>1.9.6</td>
</tr>
<tr>
<td>org.aspectj</td>
<td>aspectjtools</td>
<td>1.9.6</td>
</tr>
<tr>
<td>org.aspectj</td>
<td>aspectjweaver</td>
<td>1.9.6</td>
</tr>
<tr>
<td>org.assertj</td>
<td>assertj-core</td>
<td>3.18.1</td>
</tr>
<tr>
<td>org.awaitility</td>
<td>awaitility</td>
<td>4.0.3</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>org.awaitility</td>
<td>awaitility-groovy</td>
<td>4.0.3</td>
</tr>
<tr>
<td>org.awaitility</td>
<td>awaitility-kotlin</td>
<td>4.0.3</td>
</tr>
<tr>
<td>org.awaitility</td>
<td>awaitility-scala</td>
<td>4.0.3</td>
</tr>
<tr>
<td>org.codehaus.btm</td>
<td>btm</td>
<td>2.1.4</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-ant</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-bsf</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-cli-commons</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-cli-picocli</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-console</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-datetime</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-dateutil</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-docgenerator</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-groovyd</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-groovysh</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-jaxb</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-jmx</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-json</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-json-direct</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-jsr223</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-macro</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-nio</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-nio</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-servlet</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-servlet</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-sql</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-swing</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-templates</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-test</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-test-junit5</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-testng</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.groovy</td>
<td>groovy-xml</td>
<td>2.5.14</td>
</tr>
<tr>
<td>org.codehaus.janino</td>
<td>commons-compiler</td>
<td>3.1.2</td>
</tr>
<tr>
<td>org.codehaus.janino</td>
<td>commons-compiler-jdk</td>
<td>3.1.2</td>
</tr>
<tr>
<td>org.codehaus.janino</td>
<td>janino</td>
<td>3.1.2</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>apache-jsp</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>apache-jstl</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>infinispan-common</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>infinispan-embedded-query</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>infinispan-remote-query</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-alpn-client</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-alpn-conscrypt-client</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-alpn-conscrypt-server</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-alpn-java-client</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-alpn-java-server</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-alpn-openjdk8-client</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-alpn-openjdk8-server</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-alpn-server</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-annotations</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-ant</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-client</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-continuation</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-deploy</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-distribution</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-hazelcast</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-home</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-http</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-http-spi</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-io</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-jaas</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-jaspi</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-jmx</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-jndi</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-nosql</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-openid</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-plus</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-proxy</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-quickstart</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-reactive-httpclient</td>
<td>1.1.5</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-rewrite</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-security</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-server</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-servlet</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-servlets</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-spring</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-unixsocket</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-util</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-util-ajax</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-webapp</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty</td>
<td>jetty-xml</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.fcgi</td>
<td>fcgi-client</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.fcgi</td>
<td>fcgi-server</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.gcloud</td>
<td>jetty-gcloud-session-manager</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.http2</td>
<td>http2-client</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.http2</td>
<td>http2-common</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.http2</td>
<td>http2-hpack</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.http2</td>
<td>http2-server</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.memcached</td>
<td>jetty-memcached-sessions</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.orbit</td>
<td>javax.servlet.jsp</td>
<td>2.2.0.v201112011158</td>
</tr>
<tr>
<td>org.eclipse.jetty.osgi</td>
<td>jetty-httpservice</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.osgi</td>
<td>jetty-osgi-boot</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.osgi</td>
<td>jetty-osgi-boot-jsp</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.osgi</td>
<td>jetty-osgi-boot-warurl</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.websocket</td>
<td>javax-websocket-client-impl</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.websocket</td>
<td>javax-websocket-server-impl</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.websocket</td>
<td>websocket-api</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.websocket</td>
<td>websocket-client</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.websocket</td>
<td>websocket-common</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.websocket</td>
<td>websocket-server</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.eclipse.jetty.websocket</td>
<td>websocket-servlet</td>
<td>9.4.35.v20201120</td>
</tr>
<tr>
<td>org.ehcache</td>
<td>ehcache</td>
<td>3.9.0</td>
</tr>
<tr>
<td>org.ehcache</td>
<td>ehcache-clustered</td>
<td>3.9.0</td>
</tr>
<tr>
<td>org.ehcache</td>
<td>ehcache-transactions</td>
<td>3.9.0</td>
</tr>
<tr>
<td>org.elasticsearch</td>
<td>elasticsearch</td>
<td>7.9.3</td>
</tr>
<tr>
<td>org.elasticsearch.client</td>
<td>elasticsearch-rest-client</td>
<td>7.9.3</td>
</tr>
<tr>
<td>org.elasticsearch.client</td>
<td>elasticsearch-rest-client-sniffer</td>
<td>7.9.3</td>
</tr>
<tr>
<td>org.elasticsearch.client</td>
<td>elasticsearch-rest-high-level-client</td>
<td>7.9.3</td>
</tr>
<tr>
<td>org.elasticsearch.client</td>
<td>transport</td>
<td>7.9.3</td>
</tr>
<tr>
<td>org.elasticsearch.distribution.integ-test-zip</td>
<td>elasticsearch</td>
<td>7.9.3</td>
</tr>
<tr>
<td>org.elasticsearch.plugin</td>
<td>transport-netty4-client</td>
<td>7.9.3</td>
</tr>
<tr>
<td>org.firebirdsql.jdbc</td>
<td>jaybird-jdk17</td>
<td>3.0.9</td>
</tr>
<tr>
<td>org.firebirdsql.jdbc</td>
<td>jaybird-jdk18</td>
<td>3.0.9</td>
</tr>
<tr>
<td>org.flywaydb</td>
<td>flyway-core</td>
<td>7.1.1</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.freemarker</td>
<td>freemarker</td>
<td>2.3.30</td>
</tr>
<tr>
<td>org.glassfish</td>
<td>jakarta.el</td>
<td>3.0.3</td>
</tr>
<tr>
<td>org.glassfish.jaxb</td>
<td>codemodel</td>
<td>2.3.3</td>
</tr>
<tr>
<td>org.glassfish.jaxb</td>
<td>codemodel-annotation-compiler</td>
<td>2.3.3</td>
</tr>
<tr>
<td>org.glassfish.jaxb</td>
<td>jaxb-jxc</td>
<td>2.3.3</td>
</tr>
<tr>
<td>org.glassfish.jaxb</td>
<td>jaxb-runtime</td>
<td>2.3.3</td>
</tr>
<tr>
<td>org.glassfish.jaxb</td>
<td>jaxb-xjc</td>
<td>2.3.3</td>
</tr>
<tr>
<td>org.glassfish.jaxb</td>
<td>txw2</td>
<td>2.3.3</td>
</tr>
<tr>
<td>org.glassfish.jaxb</td>
<td>txwc2</td>
<td>2.3.3</td>
</tr>
<tr>
<td>org.glassfish.jaxb</td>
<td>xsom</td>
<td>2.3.3</td>
</tr>
<tr>
<td>org.glassfish.jersey.bundles</td>
<td>jaxrs-ri</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.connectors</td>
<td>jersey-apache-connector</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.connectors</td>
<td>jersey-grizzly-connector</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.connectors</td>
<td>jersey-helidon-connector</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.connectors</td>
<td>jersey-jdk-connector</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.connectors</td>
<td>jersey-jetty-connector</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.connectors</td>
<td>jersey-netty-connector</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers</td>
<td>jersey-container-grizzly2-http</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers</td>
<td>jersey-container-grizzly2-servlet</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers</td>
<td>jersey-container-jdk-http</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers</td>
<td>jersey-container-jetty-http</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers</td>
<td>jersey-container-jetty-servlet</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers</td>
<td>jersey-container-netty-http</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers</td>
<td>jersey-container-servlet</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers</td>
<td>jersey-container-servlet-core</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers</td>
<td>jersey-container-simple-http</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.containers.glassfish</td>
<td>jersey-gf-ejb</td>
<td>2.32</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.glassfish.jersey.core</td>
<td>jersey-client</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.core</td>
<td>jersey-common</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.core</td>
<td>jersey-server</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.core</td>
<td>jersey-bean-validation</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.core</td>
<td>jersey-declarative-linking</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.core</td>
<td>jersey-entity-filtering</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.core</td>
<td>jersey-metainf-services</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.core</td>
<td>jersey-mvc</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext</td>
<td>jersey-mvc-bean-validation</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext</td>
<td>jersey-mvc-freemarker</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext</td>
<td>jersey-mvc-jsp</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext</td>
<td>jersey-mvc-mustache</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext</td>
<td>jersey-proxy-client</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext</td>
<td>jersey-servlet-portability</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext</td>
<td>jersey-spring4</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext</td>
<td>jersey-spring5</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext</td>
<td>jersey-wadl-doclet</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.cdi</td>
<td>jersey-cdi1x</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.cdi</td>
<td>jersey-cdi1x-ban-custom-hk2-binding</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.cdi</td>
<td>jersey-cdi1x-servlet</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.cdi</td>
<td>jersey-cdi1x-transaction</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.cdi</td>
<td>jersey-cdi1x-validation</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.cdi</td>
<td>jersey-weld2-se</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.microprofile</td>
<td>jersey-mp-config</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.microprofile</td>
<td>jersey-mp-rest-client</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.rx</td>
<td>jersey-rx-client-guava</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.rx</td>
<td>jersey-rx-client-rxjava</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.ext.rx</td>
<td>jersey-rx-client-rxjava2</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.inject</td>
<td>jersey-cdi2-se</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.inject</td>
<td>jersey-hk2</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.media</td>
<td>jersey-media-jaxb</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.media</td>
<td>jersey-media-json-binding</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.media</td>
<td>jersey-media-json.jackson</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.media</td>
<td>jersey-media-json-jettison</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.media</td>
<td>jersey-media-json-processing</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.media</td>
<td>jersey-media-kryo</td>
<td>2.32</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.glassfish.jersey.media</td>
<td>jersey-media-moxy</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.media</td>
<td>jersey-media-multipart</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.media</td>
<td>jersey-media-sse</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.security</td>
<td>oauth1-client</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.security</td>
<td>oauth1-server</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.security</td>
<td>oauth1-signature</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.security</td>
<td>oauth2-client</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.test-framework</td>
<td>jersey-test-framework-core</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.test-framework</td>
<td>jersey-test-framework-util</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.test-framework.providers</td>
<td>jersey-test-framework-provider-bundle</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.test-framework.providers</td>
<td>jersey-test-framework-provider-external</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.test-framework.providers</td>
<td>jersey-test-framework-provider-grizzly2</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.test-framework.providers</td>
<td>jersey-test-framework-provider-inmemory</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.test-framework.providers</td>
<td>jersey-test-framework-provider-jdk-http</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.test-framework.providers</td>
<td>jersey-test-framework-provider-jetty</td>
<td>2.32</td>
</tr>
<tr>
<td>org.glassfish.jersey.test-framework.providers</td>
<td>jersey-test-framework-provider-simple</td>
<td>2.32</td>
</tr>
<tr>
<td>org.hamcrest</td>
<td>hamcrest</td>
<td>2.2</td>
</tr>
<tr>
<td>org.hamcrest</td>
<td>hamcrest-core</td>
<td>2.2</td>
</tr>
<tr>
<td>org.hamcrest</td>
<td>hamcrest-library</td>
<td>2.2</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-c3p0</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-core</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-ehcache</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-entitymanager</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-envers</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-hikaricp</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-java8</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-jcache</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-jpamodelgen</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-proxool</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-spatial</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-testing</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>org.hibernate</td>
<td>hibernate-vibur</td>
<td>5.4.27.Final</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>org.hibernate.validator</td>
<td>hibernate-validator</td>
<td>6.1.7.Final</td>
</tr>
<tr>
<td>org.hibernate.validator</td>
<td>hibernate-validator-annotation-processor</td>
<td>6.1.7.Final</td>
</tr>
<tr>
<td>org.hsqldb</td>
<td>hsqldb</td>
<td>2.5.1</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-anchored-keys</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-api</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-cachestore-jdbc</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-cachestore-jpa</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-cachestore-remote</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-cachestore-rest</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-cachestore-rocksdb</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-cdi-common</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-cdi-embedded</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-cdi-remote</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-checkstyle</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-cli-client</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-client-hotrod</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-client-rest</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-clustered-counter</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-clustered-lock</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-commons</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-commons-test</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-component-annotations</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-component-processor</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-console</td>
<td>0.7.2.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-core</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-extended-statistics</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-hibernate-cache-commons</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-hibernate-cache-spi</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-hibernate-cache-v51</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-hibernate-cache-v53</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-jboss-marshalling</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-jcache</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-jcache-commons</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-jcache-remote</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-key-value-store-client</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-marshaller-kryo</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-marshaller-kryo-bundle</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-marshaller-protostuff</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-marshaller-protostuff-bundle</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-multimap</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-objectfilter</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-osgi</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-persistence-soft-index</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-query</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-query-core</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-query-dsl</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-remote-query-client</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-remote-query-server</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-scripting</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-server-core</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-server-hotrod</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-server-memcached</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-server-rest</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-server-router</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-server-runtime</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-spring5-common</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-spring5-embedded</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-spring5-remote</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-tasks</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-tasks-api</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan</td>
<td>infinispan-tools</td>
<td>11.0.8.Final</td>
</tr>
<tr>
<td>org.infinispan.protostream</td>
<td>protostream</td>
<td>4.3.4.Final</td>
</tr>
<tr>
<td>org.infinispan.protostream</td>
<td>protostream-processor</td>
<td>4.3.4.Final</td>
</tr>
<tr>
<td>org.influxdb</td>
<td>influxdb-java</td>
<td>2.20</td>
</tr>
<tr>
<td>org.jboss</td>
<td>jboss-transaction-spi</td>
<td>7.6.0.Final</td>
</tr>
<tr>
<td>org.jboss.logging</td>
<td>jboss-logging</td>
<td>3.4.1.Final</td>
</tr>
<tr>
<td>org.jdom</td>
<td>jdom2</td>
<td>2.0.6</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-compiler</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-compiler-embeddable</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-daemon-client</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-main-kts</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-osgi-bundle</td>
<td>1.4.21</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-reflect</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-script-runtime</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-script-util</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-scripting-common</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-scripting-ide-services</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-scripting-jvm</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-scripting-jvm-host</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-stdlib</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-stdlib-common</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-stdlib-jdk7</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-stdlib-jdk8</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-stdlib-js</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-test</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-test-annotations-common</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-test-common</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-test-js</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-test-junit</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-test-junit5</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlin-test-testng</td>
<td>1.4.21</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-android</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-core</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-core-jvm</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-debug</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-guava</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-javafx</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-jdk8</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-jdk9</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-play-services</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-reactive</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-reactor</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-rx2</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-rx3</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-slf4j</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-swing</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jetbrains.kotlin</td>
<td>kotlinx-coroutines-test</td>
<td>1.4.2</td>
</tr>
<tr>
<td>org.jolokia</td>
<td>jolokia-core</td>
<td>1.6.2</td>
</tr>
<tr>
<td>org.jooq</td>
<td>jooq</td>
<td>3.14.4</td>
</tr>
<tr>
<td>org.jooq</td>
<td>jooq-codegen</td>
<td>3.14.4</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>org.jooq</td>
<td>jooq-kotlin</td>
<td>3.14.4</td>
</tr>
<tr>
<td>org.jooq</td>
<td>jooq-meta</td>
<td>3.14.4</td>
</tr>
<tr>
<td>org.junit.jupiter</td>
<td>junit-jupiter</td>
<td>5.7.0</td>
</tr>
<tr>
<td>org.junit.jupiter</td>
<td>junit-jupiter-api</td>
<td>5.7.0</td>
</tr>
<tr>
<td>org.junit.jupiter</td>
<td>junit-jupiter-engine</td>
<td>5.7.0</td>
</tr>
<tr>
<td>org.junit.jupiter</td>
<td>junit-jupiter-migrationsupport</td>
<td>5.7.0</td>
</tr>
<tr>
<td>org.junit.jupiter</td>
<td>junit-jupiter-params</td>
<td>5.7.0</td>
</tr>
<tr>
<td>org.junit.platform</td>
<td>junit-platform-commons</td>
<td>1.7.0</td>
</tr>
<tr>
<td>org.junit.platform</td>
<td>junit-platform-console</td>
<td>1.7.0</td>
</tr>
<tr>
<td>org.junit.platform</td>
<td>junit-platform-engine</td>
<td>1.7.0</td>
</tr>
<tr>
<td>org.junit.platform</td>
<td>junit-platform-jfr</td>
<td>1.7.0</td>
</tr>
<tr>
<td>org.junit.platform</td>
<td>junit-platform-launcher</td>
<td>1.7.0</td>
</tr>
<tr>
<td>org.junit.platform</td>
<td>junit-platform-reporting</td>
<td>1.7.0</td>
</tr>
<tr>
<td>org.junit.platform</td>
<td>junit-platform-runner</td>
<td>1.7.0</td>
</tr>
<tr>
<td>org.junit.platform</td>
<td>junit-platform-suite-api</td>
<td>1.7.0</td>
</tr>
<tr>
<td>org.junit.platform</td>
<td>junit-platform-testkit</td>
<td>1.7.0</td>
</tr>
<tr>
<td>org.junit.vintage</td>
<td>junit-vintage-engine</td>
<td>5.7.0</td>
</tr>
<tr>
<td>org.jvnet.mimepull</td>
<td>mimepull</td>
<td>1.9.13</td>
</tr>
<tr>
<td>org.liquibase</td>
<td>liquibase-core</td>
<td>3.10.3</td>
</tr>
<tr>
<td>org.mariadb</td>
<td>r2dbc-mariadb</td>
<td>0.8.4-rc</td>
</tr>
<tr>
<td>org.mariadb.jdbc</td>
<td>mariadb-java-client</td>
<td>2.7.1</td>
</tr>
<tr>
<td>org.messaginghub</td>
<td>pooled-jms</td>
<td>1.2.1</td>
</tr>
<tr>
<td>org.mockito</td>
<td>mockito-core</td>
<td>3.6.28</td>
</tr>
<tr>
<td>org.mockito</td>
<td>mockito-inline</td>
<td>3.6.28</td>
</tr>
<tr>
<td>org.mockito</td>
<td>mockito-junit-jupiter</td>
<td>3.6.28</td>
</tr>
<tr>
<td>org.mongodb</td>
<td>bson</td>
<td>4.1.1</td>
</tr>
<tr>
<td>org.mongodb</td>
<td>mongodb-driver-core</td>
<td>4.1.1</td>
</tr>
<tr>
<td>org.mongodb</td>
<td>mongodb-driver-legacy</td>
<td>4.1.1</td>
</tr>
<tr>
<td>org.mongodb</td>
<td>mongodb-driver-reactive-streams</td>
<td>4.1.1</td>
</tr>
<tr>
<td>org.mongodb</td>
<td>mongodb-driver-sync</td>
<td>4.1.1</td>
</tr>
<tr>
<td>org.mortbay.jasper</td>
<td>apache-el</td>
<td>8.5.54</td>
</tr>
<tr>
<td>org.neo4j.driver</td>
<td>neo4j-java-driver</td>
<td>4.1.1</td>
</tr>
<tr>
<td>org.postgresql</td>
<td>postgresql</td>
<td>42.2.18</td>
</tr>
<tr>
<td>org.projectlombok</td>
<td>lombok</td>
<td>1.18.16</td>
</tr>
<tr>
<td>org.quartz-scheduler</td>
<td>quartz</td>
<td>2.3.2</td>
</tr>
<tr>
<td>org.quartz-scheduler</td>
<td>quartz-jobs</td>
<td>2.3.2</td>
</tr>
<tr>
<td>org.reactivestreams</td>
<td>reactive-streams</td>
<td>1.0.3</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>htmlunit-driver</td>
<td>2.44.0</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------</td>
<td>----------</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-api</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-chrome-driver</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-edge-driver</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-firefox-driver</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-ie-driver</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-java</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-opera-driver</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-remote-driver</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-safari-driver</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.seleniumhq.selenium</td>
<td>selenium-support</td>
<td>3.141.59</td>
</tr>
<tr>
<td>org.skyscreamer</td>
<td>jsonassert</td>
<td>1.5.0</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>jcl-over-slf4j</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>jul-to-slf4j</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>log4j-over-slf4j</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>slf4j-api</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>slf4j-ext</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>slf4j-jcl</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>slf4j-jdk14</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>slf4j-log4j12</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>slf4j-nop</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.slf4j</td>
<td>slf4j-simple</td>
<td>1.7.30</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-aop</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-aspects</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-beans</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-context</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-context-indexer</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-context-support</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-core</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-expression</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-instrument</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-jcl</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-jdbc</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-jms</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-messaging</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-orm</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-oxm</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-r2dbc</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-test</td>
<td>5.3.3</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-tx</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-web</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-webflux</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-webmvc</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework</td>
<td>spring-websocket</td>
<td>5.3.3</td>
</tr>
<tr>
<td>org.springframework.amqp</td>
<td>spring-amqp</td>
<td>2.3.4</td>
</tr>
<tr>
<td>org.springframework.amqp</td>
<td>spring-rabbit</td>
<td>2.3.4</td>
</tr>
<tr>
<td>org.springframework.amqp</td>
<td>spring-rabbit-junit</td>
<td>2.3.4</td>
</tr>
<tr>
<td>org.springframework.amqp</td>
<td>spring-rabbit-test</td>
<td>2.3.4</td>
</tr>
<tr>
<td>org.springframework.batch</td>
<td>spring-batch-core</td>
<td>4.3.1</td>
</tr>
<tr>
<td>org.springframework.batch</td>
<td>spring-batch-infrastructure</td>
<td>4.3.1</td>
</tr>
<tr>
<td>org.springframework.batch</td>
<td>spring-batch-integration</td>
<td>4.3.1</td>
</tr>
<tr>
<td>org.springframework.batch</td>
<td>spring-batch-test</td>
<td>4.3.1</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-actuator</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-actuator-autoconfigure</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-autoconfigure</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-autoconfigure-processor</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-buildpack-platform</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-configuration-metadata</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-configuration-processor</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-devtools</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-jarmode-layertools</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-loader</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-loader-tools</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-properties-migrator</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-activemq</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-actuator</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-amqp</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-aop</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-artemis</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-batch</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-cache</td>
<td>2.4.2</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-cassandra</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-cassandra-reactive</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-couchbase</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-couchbase-reactive</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-elasticsearch</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-jdbc</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-jpa</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-ldap</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-mongodb</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-mongodb-reactive</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-neo4j</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-r2dbc</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-redis</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-redis-reactive</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-rest</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-data-solr</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-freemarker</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-groovy-templates</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-hateoas</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-integration</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-jdbc</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-jersey</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-jetty</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-jooq</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-json</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-jta-atomikos</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-jta-bitronix</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-log4j2</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-logging</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-mail</td>
<td>2.4.2</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-mustache</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-oauth2-client</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-oauth2-resource-server</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-quartz</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-reactor-netty</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-rsocket</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-security</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-test</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-thymeleaf</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-tomcat</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-undertow</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-validation</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-web</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-web-services</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-webflux</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-starter-websocket</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-test</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.boot</td>
<td>spring-boot-test-autoconfigure</td>
<td>2.4.2</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-cassandra</td>
<td>3.1.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-commons</td>
<td>2.4.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-couchbase</td>
<td>4.1.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-elasticsearch</td>
<td>4.1.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-envers</td>
<td>2.4.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-geode</td>
<td>2.4.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-jdbc</td>
<td>2.1.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-jpa</td>
<td>2.4.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-keyvalue</td>
<td>2.4.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-ldap</td>
<td>2.4.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-mongodb</td>
<td>3.1.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-neo4j</td>
<td>6.0.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-r2dbc</td>
<td>1.2.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-redis</td>
<td>2.4.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-relational</td>
<td>2.1.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-rest-core</td>
<td>3.4.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-rest-hal-explorer</td>
<td>3.4.3</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-rest-webmvc</td>
<td>3.4.3</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.springframework.data</td>
<td>spring-data-solr</td>
<td>4.3.3</td>
</tr>
<tr>
<td>org.springframework.hateoas</td>
<td>spring-hateoas</td>
<td>1.2.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-amqp</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-core</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-event</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-feed</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-file</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-ftp</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-gemfire</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-groovy</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-http</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-ip</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-jdbc</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-jms</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-jmx</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-jpa</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-kafka</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-mail</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-mongodb</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-mqtt</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-r2dbc</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-redis</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-rmi</td>
<td>5.4.3</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-rsocket</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-scripting</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-security</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-sftp</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-stomp</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-stream</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-syslog</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-test</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-test-support</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-webflux</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-websocket</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-ws</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-xml</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-xmpp</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.integration</td>
<td>spring-integration-zookeeper</td>
<td>5.4.3</td>
</tr>
<tr>
<td>org.springframework.kafka</td>
<td>spring-kafka</td>
<td>2.6.5</td>
</tr>
<tr>
<td>org.springframework.kafka</td>
<td>spring-kafka-test</td>
<td>2.6.5</td>
</tr>
<tr>
<td>org.springframework.ldap</td>
<td>spring-ldap-core</td>
<td>2.3.3.RELEASE</td>
</tr>
<tr>
<td>org.springframework.ldap</td>
<td>spring-ldap-core-tiger</td>
<td>2.3.3.RELEASE</td>
</tr>
<tr>
<td>org.springframework.ldap</td>
<td>spring-ldap-ldif-batch</td>
<td>2.3.3.RELEASE</td>
</tr>
<tr>
<td>org.springframework.ldap</td>
<td>spring-ldap-ldif-core</td>
<td>2.3.3.RELEASE</td>
</tr>
<tr>
<td>org.springframework.ldap</td>
<td>spring-ldap-odm</td>
<td>2.3.3.RELEASE</td>
</tr>
<tr>
<td>org.springframework.ldap</td>
<td>spring-ldap-test</td>
<td>2.3.3.RELEASE</td>
</tr>
<tr>
<td>org.springframework.restdocs</td>
<td>spring-restdocs-asciidocor</td>
<td>2.0.5.RELEASE</td>
</tr>
<tr>
<td>org.springframework.restdocs</td>
<td>spring-restdocs-core</td>
<td>2.0.5.RELEASE</td>
</tr>
<tr>
<td>org.springframework.restdocs</td>
<td>spring-restdocs-mockmvc</td>
<td>2.0.5.RELEASE</td>
</tr>
<tr>
<td>Group ID</td>
<td>Artifact ID</td>
<td>Version</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>org.springframework.restdocs</td>
<td>spring-restdocs-restassured</td>
<td>2.0.5.RELEASE</td>
</tr>
<tr>
<td>org.springframework.restdocs</td>
<td>spring-restdocs-webtestclient</td>
<td>2.0.5.RELEASE</td>
</tr>
<tr>
<td>org.springframework.retry</td>
<td>spring-retry</td>
<td>1.3.1</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-acl</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-aspects</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-cas</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-config</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-core</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-crypto</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-data</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-ldap</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-messaging</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-oauth2-client</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-oauth2-core</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-oauth2-jose</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-oauth2-resource-server</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-openid</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-remoting</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-rsocket</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-saml2-service-provider</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-taglibs</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-test</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.security</td>
<td>spring-security-web</td>
<td>5.4.2</td>
</tr>
<tr>
<td>org.springframework.session</td>
<td>spring-session-core</td>
<td>2.4.1</td>
</tr>
<tr>
<td>org.springframework.session</td>
<td>spring-session-data-geode</td>
<td>2.4.0</td>
</tr>
<tr>
<td>org.springframework.session</td>
<td>spring-session-data-mongodb</td>
<td>2.4.1</td>
</tr>
<tr>
<td>org.springframework.session</td>
<td>spring-session-data-redis</td>
<td>2.4.1</td>
</tr>
<tr>
<td>org.springframework.session</td>
<td>spring-session-hazelcast</td>
<td>2.4.1</td>
</tr>
<tr>
<td>org.springframework.session</td>
<td>spring-session-jdbc</td>
<td>2.4.1</td>
</tr>
<tr>
<td>org.springframework.ws</td>
<td>spring-ws-core</td>
<td>3.0.10.RELEASE</td>
</tr>
<tr>
<td>org.springframework.ws</td>
<td>spring-ws-security</td>
<td>3.0.10.RELEASE</td>
</tr>
<tr>
<td>org.springframework.ws</td>
<td>spring-ws-support</td>
<td>3.0.10.RELEASE</td>
</tr>
<tr>
<td>org.springframework.ws</td>
<td>spring-ws-test</td>
<td>3.0.10.RELEASE</td>
</tr>
<tr>
<td>org.springframework.ws</td>
<td>spring-xml</td>
<td>3.0.10.RELEASE</td>
</tr>
<tr>
<td>org.thymeleaf</td>
<td>thymeleaf</td>
<td>3.0.12.RELEASE</td>
</tr>
<tr>
<td>org.thymeleaf</td>
<td>thymeleaf-spring5</td>
<td>3.0.12.RELEASE</td>
</tr>
<tr>
<td>org.thymeleaf.extras</td>
<td>thymeleaf-extras-java8time</td>
<td>3.0.4.RELEASE</td>
</tr>
</tbody>
</table>
### 10.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.

<table>
<thead>
<tr>
<th>Library</th>
<th>Version Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveMQ</td>
<td>activemq.version</td>
</tr>
<tr>
<td>ANTLR2</td>
<td>antlr2.version</td>
</tr>
<tr>
<td>AppEngine SDK</td>
<td>appengine-sdk.version</td>
</tr>
<tr>
<td>Artemis</td>
<td>artemis.version</td>
</tr>
<tr>
<td>AspectJ</td>
<td>aspectj.version</td>
</tr>
<tr>
<td>AssertJ</td>
<td>assertj.version</td>
</tr>
<tr>
<td>Atomikos</td>
<td>atomikos.version</td>
</tr>
<tr>
<td>Awaitility</td>
<td>awaitility.version</td>
</tr>
<tr>
<td>Bitronix</td>
<td>bitronix.version</td>
</tr>
<tr>
<td>Build Helper Maven Plugin</td>
<td>build-helper-maven-plugin.version</td>
</tr>
<tr>
<td>Byte Buddy</td>
<td>byte-buddy.version</td>
</tr>
<tr>
<td>Caffeine</td>
<td>caffeine.version</td>
</tr>
<tr>
<td>Cassandra Driver</td>
<td>cassandra-driver.version</td>
</tr>
<tr>
<td>Classmate</td>
<td>classmate.version</td>
</tr>
<tr>
<td>Commons Codec</td>
<td>commons-codec.version</td>
</tr>
<tr>
<td>Commons DBCP2</td>
<td>commons-dbcp2.version</td>
</tr>
<tr>
<td>Commons Lang3</td>
<td>commons-lang3.version</td>
</tr>
<tr>
<td>Commons Pool</td>
<td>commons-pool.version</td>
</tr>
<tr>
<td>Commons Pool2</td>
<td>commons-pool2.version</td>
</tr>
<tr>
<td>Couchbase Client</td>
<td>couchbase-client.version</td>
</tr>
<tr>
<td>Library</td>
<td>Version Property</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>DB2 JDBC</td>
<td>db2-jdbc.version</td>
</tr>
<tr>
<td>Dependency Management Plugin</td>
<td>dependency-management-plugin.version</td>
</tr>
<tr>
<td>Derby</td>
<td>derby.version</td>
</tr>
<tr>
<td>Dropwizard Metrics</td>
<td>dropwizard-metrics.version</td>
</tr>
<tr>
<td>Ehcache</td>
<td>ehcache.version</td>
</tr>
<tr>
<td>Ehcache3</td>
<td>ehcache3.version</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>elasticsearch.version</td>
</tr>
<tr>
<td>Embedded Mongo</td>
<td>embedded-mongo.version</td>
</tr>
<tr>
<td>Flyway</td>
<td>flyway.version</td>
</tr>
<tr>
<td>FreeMarker</td>
<td>freemarker.version</td>
</tr>
<tr>
<td>Git Commit ID Plugin</td>
<td>git-commit-id-plugin.version</td>
</tr>
<tr>
<td>Glassfish EL</td>
<td>glassfish-el.version</td>
</tr>
<tr>
<td>Glassfish JAXB</td>
<td>glassfish-jaxb.version</td>
</tr>
<tr>
<td>Groovy</td>
<td>groovy.version</td>
</tr>
<tr>
<td>Gson</td>
<td>gson.version</td>
</tr>
<tr>
<td>H2</td>
<td>h2.version</td>
</tr>
<tr>
<td>Hamcrest</td>
<td>hamcrest.version</td>
</tr>
<tr>
<td>Hazelcast</td>
<td>hazelcast.version</td>
</tr>
<tr>
<td>Hazelcast Hibernate5</td>
<td>hazelcast-hibernate5.version</td>
</tr>
<tr>
<td>Hibernate</td>
<td>hibernate.version</td>
</tr>
<tr>
<td>Hibernate Validator</td>
<td>hibernate-validator.version</td>
</tr>
<tr>
<td>HikariCP</td>
<td>hikaricp.version</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>hsqldb.version</td>
</tr>
<tr>
<td>HtmlUnit</td>
<td>htmlunit.version</td>
</tr>
<tr>
<td>HttpAsyncClient</td>
<td>httpasyncclient.version</td>
</tr>
<tr>
<td>HttpClient</td>
<td>httpclient.version</td>
</tr>
<tr>
<td>HttpCore</td>
<td>httpcore.version</td>
</tr>
<tr>
<td>Infinispan</td>
<td>infinispan.version</td>
</tr>
<tr>
<td>InfluxDB Java</td>
<td>influxdb-java.version</td>
</tr>
<tr>
<td>Jackson Bom</td>
<td>jackson-bom.version</td>
</tr>
<tr>
<td>Jakarta Activation</td>
<td>jakarta-activation.version</td>
</tr>
<tr>
<td>Jakarta Annotation</td>
<td>jakarta-annotation.version</td>
</tr>
<tr>
<td>Jakarta JMS</td>
<td>jakarta-jms.version</td>
</tr>
<tr>
<td>Jakarta Json</td>
<td>jakarta-json.version</td>
</tr>
<tr>
<td>Jakarta Json Bind</td>
<td>jakarta-json-bind.version</td>
</tr>
<tr>
<td>Jakarta Mail</td>
<td>jakarta-mail.version</td>
</tr>
<tr>
<td>Jakarta Persistence</td>
<td>jakarta-persistence.version</td>
</tr>
<tr>
<td>Jakarta Servlet</td>
<td>jakarta-servlet.version</td>
</tr>
<tr>
<td>Library</td>
<td>Version Property</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>Jakarta Servlet JSP JSTL</td>
<td>jakarta-servlet-jsp-jstl.version</td>
</tr>
<tr>
<td>Jakarta Transaction</td>
<td>jakarta-transaction.version</td>
</tr>
<tr>
<td>Jakarta Validation</td>
<td>jakarta-validation.version</td>
</tr>
<tr>
<td>Jakarta WebSocket</td>
<td>jakarta-websocket.version</td>
</tr>
<tr>
<td>Jakarta WS RS</td>
<td>jakarta-ws-rs.version</td>
</tr>
<tr>
<td>Jakarta XML Bind</td>
<td>jakarta-xml-bind.version</td>
</tr>
<tr>
<td>Jakarta XML SOAP</td>
<td>jakarta-xml-soap.version</td>
</tr>
<tr>
<td>Jakarta XML WS</td>
<td>jakarta-xml-ws.version</td>
</tr>
<tr>
<td>Janino</td>
<td>janino.version</td>
</tr>
<tr>
<td>Javax Activation</td>
<td>javax-activation.version</td>
</tr>
<tr>
<td>Javax Annotation</td>
<td>javax-annotation.version</td>
</tr>
<tr>
<td>Javax Cache</td>
<td>javax-cache.version</td>
</tr>
<tr>
<td>Javax JAXB</td>
<td>javax-jaxb.version</td>
</tr>
<tr>
<td>Javax JAXWS</td>
<td>javax-jaxws.version</td>
</tr>
<tr>
<td>Javax JMS</td>
<td>javax-jms.version</td>
</tr>
<tr>
<td>Javax Json</td>
<td>javax-json.version</td>
</tr>
<tr>
<td>Javax JsonB</td>
<td>javax-jsonb.version</td>
</tr>
<tr>
<td>Javax Mail</td>
<td>javax-mail.version</td>
</tr>
<tr>
<td>Javax Money</td>
<td>javax-money.version</td>
</tr>
<tr>
<td>Javax Persistence</td>
<td>javax-persistence.version</td>
</tr>
<tr>
<td>Javax Transaction</td>
<td>javaxtransaction.version</td>
</tr>
<tr>
<td>Javax Validation</td>
<td>javax-validation.version</td>
</tr>
<tr>
<td>Javax WebSocket</td>
<td>javax-websocket.version</td>
</tr>
<tr>
<td>Jaxen</td>
<td>jaxen.version</td>
</tr>
<tr>
<td>Jaybird</td>
<td>jaybird.version</td>
</tr>
<tr>
<td>JBoss Logging</td>
<td>jboss-logging.version</td>
</tr>
<tr>
<td>JBoss Transaction SPI</td>
<td>jboss-transaction-spi.version</td>
</tr>
<tr>
<td>JDOM2</td>
<td>jdom2.version</td>
</tr>
<tr>
<td>Jedis</td>
<td>jedis.version</td>
</tr>
<tr>
<td>Jersey</td>
<td>jersey.version</td>
</tr>
<tr>
<td>Jetty</td>
<td>jetty.version</td>
</tr>
<tr>
<td>Jetty EL</td>
<td>jetty-el.version</td>
</tr>
<tr>
<td>Jetty JSP</td>
<td>jetty-jsp.version</td>
</tr>
<tr>
<td>Jetty Reactive HTTPClient</td>
<td>jetty-reactive-httpclient.version</td>
</tr>
<tr>
<td>JMustache HTTPClient</td>
<td>jmustache.version</td>
</tr>
<tr>
<td>Johnzon</td>
<td>johnzon.version</td>
</tr>
<tr>
<td>Jolokia</td>
<td>jolokia.version</td>
</tr>
<tr>
<td>jOOQ</td>
<td>jooq.version</td>
</tr>
</tbody>
</table>

622
<table>
<thead>
<tr>
<th>Library</th>
<th>Version Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Json Path</td>
<td>json-path.version</td>
</tr>
<tr>
<td>Json-smart</td>
<td>json-smart.version</td>
</tr>
<tr>
<td>JsonAssert</td>
<td>jsonassert.version</td>
</tr>
<tr>
<td>JSTL</td>
<td>jstl.version</td>
</tr>
<tr>
<td>JTDS</td>
<td>jtds.version</td>
</tr>
<tr>
<td>JUnit</td>
<td>junit.version</td>
</tr>
<tr>
<td>JUnit Jupiter</td>
<td>junit-jupiter.version</td>
</tr>
<tr>
<td>Kafka</td>
<td>kafka.version</td>
</tr>
<tr>
<td>Kotlin</td>
<td>kotlin.version</td>
</tr>
<tr>
<td>Kotlin Coroutines</td>
<td>kotlin-coroutines.version</td>
</tr>
<tr>
<td>Lettuce</td>
<td>lettuce.version</td>
</tr>
<tr>
<td>Liquibase</td>
<td>liquibase.version</td>
</tr>
<tr>
<td>Log4j2</td>
<td>log4j2.version</td>
</tr>
<tr>
<td>Logback</td>
<td>logback.version</td>
</tr>
<tr>
<td>Lombok</td>
<td>lombok.version</td>
</tr>
<tr>
<td>MariaDB</td>
<td>mariadb.version</td>
</tr>
<tr>
<td>Maven AntRun Plugin</td>
<td>maven-antrun-plugin.version</td>
</tr>
<tr>
<td>Maven Assembly Plugin</td>
<td>maven-assembly-plugin.version</td>
</tr>
<tr>
<td>Maven Clean Plugin</td>
<td>maven-clean-plugin.version</td>
</tr>
<tr>
<td>Maven Compiler Plugin</td>
<td>maven-compiler-plugin.version</td>
</tr>
<tr>
<td>Maven Dependency Plugin</td>
<td>maven-dependency-plugin.version</td>
</tr>
<tr>
<td>Maven Deploy Plugin</td>
<td>maven-deploy-plugin.version</td>
</tr>
<tr>
<td>Maven Enforcer Plugin</td>
<td>maven-enforcer-plugin.version</td>
</tr>
<tr>
<td>Maven Failsafe Plugin</td>
<td>maven-failsafe-plugin.version</td>
</tr>
<tr>
<td>Maven Help Plugin</td>
<td>maven-help-plugin.version</td>
</tr>
<tr>
<td>Maven Install Plugin</td>
<td>maven-install-plugin.version</td>
</tr>
<tr>
<td>Maven Invoker Plugin</td>
<td>maven-invoker-plugin.version</td>
</tr>
<tr>
<td>Maven Jar Plugin</td>
<td>maven-jar-plugin.version</td>
</tr>
<tr>
<td>Maven Javadoc Plugin</td>
<td>maven-javadoc-plugin.version</td>
</tr>
<tr>
<td>Maven Resources Plugin</td>
<td>maven-resources-plugin.version</td>
</tr>
<tr>
<td>Maven Shade Plugin</td>
<td>maven-shade-plugin.version</td>
</tr>
<tr>
<td>Maven Source Plugin</td>
<td>maven-source-plugin.version</td>
</tr>
<tr>
<td>Maven Surefire Plugin</td>
<td>maven-surefire-plugin.version</td>
</tr>
<tr>
<td>Maven War Plugin</td>
<td>maven-war-plugin.version</td>
</tr>
<tr>
<td>Micrometer</td>
<td>micrometer.version</td>
</tr>
<tr>
<td>MIMEPull</td>
<td>mimepull.version</td>
</tr>
<tr>
<td>Mockito</td>
<td>mockito.version</td>
</tr>
<tr>
<td>MongoDB</td>
<td>mongodb.version</td>
</tr>
<tr>
<td>Library</td>
<td>Version Property</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>MSSQL JDBC</td>
<td>mssql-jdbc.version</td>
</tr>
<tr>
<td>MySQL</td>
<td>mysql.version</td>
</tr>
<tr>
<td>NekoHTML</td>
<td>nekohtml.version</td>
</tr>
<tr>
<td>Neo4j Java Driver</td>
<td>neo4j-java-driver.version</td>
</tr>
<tr>
<td>Netty</td>
<td>netty.version</td>
</tr>
<tr>
<td>Netty tcNative</td>
<td>netty-tcnative.version</td>
</tr>
<tr>
<td>Nimbus JOSE JWT</td>
<td>nimbus-jose-jwt.version</td>
</tr>
<tr>
<td>OAuth2 OIDC SDK</td>
<td>oauth2-oidc-sdk.version</td>
</tr>
<tr>
<td>OJDBC</td>
<td>ojdbc.version</td>
</tr>
<tr>
<td>OkHttp3</td>
<td>okhttp3.version</td>
</tr>
<tr>
<td>Oracle Database</td>
<td>oracle-database.version</td>
</tr>
<tr>
<td>Pooled JMS</td>
<td>pooled-jms.version</td>
</tr>
<tr>
<td>Postgresql</td>
<td>postgresql.version</td>
</tr>
<tr>
<td>Prometheus PushGateway</td>
<td>prometheus-pushgateway.version</td>
</tr>
<tr>
<td>Quartz</td>
<td>quartz.version</td>
</tr>
<tr>
<td>QueryDSL</td>
<td>querydsl.version</td>
</tr>
<tr>
<td>R2DBC Bom</td>
<td>r2dbc-bom.version</td>
</tr>
<tr>
<td>Rabbit AMQP Client</td>
<td>rabbit-amqp-client.version</td>
</tr>
<tr>
<td>Reactive Streams</td>
<td>reactive-streams.version</td>
</tr>
<tr>
<td>Reactor Bom</td>
<td>reactor-bom.version</td>
</tr>
<tr>
<td>REST Assured</td>
<td>rest-assured.version</td>
</tr>
<tr>
<td>RSocket</td>
<td>rsocket.version</td>
</tr>
<tr>
<td>RxJava</td>
<td>rxjava.version</td>
</tr>
<tr>
<td>RxJava Adapter</td>
<td>rxjava-adapter.version</td>
</tr>
<tr>
<td>RxJava2</td>
<td>rxjava2.version</td>
</tr>
<tr>
<td>SAAJ Impl</td>
<td>saaj-impl.version</td>
</tr>
<tr>
<td>Selenium</td>
<td>selenium.version</td>
</tr>
<tr>
<td>Selenium HtmlUnit</td>
<td>selenium-htmlunit.version</td>
</tr>
<tr>
<td>SendGrid</td>
<td>sendgrid.version</td>
</tr>
<tr>
<td>Servlet API</td>
<td>servlet-api.version</td>
</tr>
<tr>
<td>SLF4J</td>
<td>slf4j.version</td>
</tr>
<tr>
<td>SnakeYAML</td>
<td>snakeyaml.version</td>
</tr>
<tr>
<td>Solr</td>
<td>solr.version</td>
</tr>
<tr>
<td>Spring AMQP</td>
<td>spring-amqp.version</td>
</tr>
<tr>
<td>Spring Batch</td>
<td>spring-batch.version</td>
</tr>
<tr>
<td>Spring Data Bom</td>
<td>spring-data-bom.version</td>
</tr>
<tr>
<td>Spring Framework</td>
<td>spring-framework.version</td>
</tr>
<tr>
<td>Spring HATEOAS</td>
<td>spring-hateoas.version</td>
</tr>
<tr>
<td>Library</td>
<td>Version Property</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Spring Integration</td>
<td>spring-integration.version</td>
</tr>
<tr>
<td>Spring Kafka</td>
<td>spring-kafka.version</td>
</tr>
<tr>
<td>Spring LDAP</td>
<td>spring-ldap.version</td>
</tr>
<tr>
<td>Spring RESTDocs</td>
<td>spring-restdocs.version</td>
</tr>
<tr>
<td>Spring Retry</td>
<td>spring-retry.version</td>
</tr>
<tr>
<td>Spring Security</td>
<td>spring-security.version</td>
</tr>
<tr>
<td>Spring Session Bom</td>
<td>spring-session-bom.version</td>
</tr>
<tr>
<td>Spring WS</td>
<td>spring-ws.version</td>
</tr>
<tr>
<td>SQLite JDBC</td>
<td>sqlite-jdbc.version</td>
</tr>
<tr>
<td>Sun Mail</td>
<td>sun-mail.version</td>
</tr>
<tr>
<td>Thymeleaf</td>
<td>thymeleaf.version</td>
</tr>
<tr>
<td>Thymeleaf Extras Data Attribute</td>
<td>thymeleaf-extras-data-attribute.version</td>
</tr>
<tr>
<td>Thymeleaf Extras Java8Time</td>
<td>thymeleaf-extras-java8time.version</td>
</tr>
<tr>
<td>Thymeleaf Extras SpringSecurity</td>
<td>thymeleaf-extras-springsecurity.version</td>
</tr>
<tr>
<td>Thymeleaf Layout Dialect</td>
<td>thymeleaf-layout-dialect.version</td>
</tr>
<tr>
<td>Tomcat</td>
<td>tomcat.version</td>
</tr>
<tr>
<td>UnboundID LDAPSDK</td>
<td>unboundid-ldapsdk.version</td>
</tr>
<tr>
<td>Undertow</td>
<td>undertow.version</td>
</tr>
<tr>
<td>Versions Maven Plugin</td>
<td>versions-maven-plugin.version</td>
</tr>
<tr>
<td>WebJars HAL Browser</td>
<td>webjars-hal-browser.version</td>
</tr>
<tr>
<td>WebJars Locator Core</td>
<td>webjars-locator-core.version</td>
</tr>
<tr>
<td>WSDL4j</td>
<td>wsdl4j.version</td>
</tr>
<tr>
<td>XML Maven Plugin</td>
<td>xml-maven-plugin.version</td>
</tr>
<tr>
<td>XmlUnit2</td>
<td>xmlunit2.version</td>
</tr>
</tbody>
</table>