Spring Cloud

Table of Contents

1. Features . . . 2
2. Release Train Versions . . . 3
Spring Cloud AWS . . . 4
  3. Using Amazon Web Services . . . 5
  4. Basic setup . . . 6
    4.1. Spring Cloud AWS maven dependency management . . . 6
    4.2. Amazon SDK dependency version management . . . 7
    4.3. Amazon SDK configuration . . . 7
  5. Cloud environment . . . 14
    5.1. Retrieving instance metadata . . . 14
    5.2. Integrating your Spring Cloud application with the AWS Parameter Store . . . 19
    5.3. Integrating your Spring Cloud application with the AWS Secrets Manager . . . 21
  6. Managing cloud environments . . . 23
    6.1. Automatic CloudFormation configuration . . . 23
    6.2. Manual CloudFormation configuration . . . 24
    6.3. CloudFormation configuration with Java config classes . . . 24
    6.4. CloudFormation configuration in Spring Boot . . . 25
    6.5. Manual name resolution . . . 25
    6.6. Stack Tags . . . 26
    6.7. Using custom CloudFormation client . . . 26
  7. Messaging . . . 27
    7.1. Configuring messaging . . . 27
    7.2. SQS support . . . 27
    7.3. SNS support . . . 33
    7.4. Using CloudFormation . . . 36
  8. Caching . . . 37
    8.1. Configuring dependencies for Redis caches . . . 37
    8.2. Configuring caching with XML . . . 37
    8.3. Configuring caching using Java configuration . . . 39
    8.4. Configuring caching in Spring Boot . . . 39
    8.5. Using caching . . . 40
    8.6. Memcached client implementation . . . 40
    8.7. Using CloudFormation . . . 40
  9. Data Access with JDBC . . . 42
    9.1. Configuring data source . . . 42
    9.2. Configuring data source with Java config . . . 44
    9.3. Configuring data source in Spring Boot . . . 45
    9.4. Read-replica configuration . . . 46

9.5. Failover support 47
9.6. CloudFormation support 48
9.7. Database tags 49
10. Sending mails 51
10.1. Configuring the mail sender 51
10.2. Sending simple mails 51
10.3. Sending attachments 52
10.4. Configuring regions 53
10.5. Authenticating e-mails 53
11. Resource handling 55
11.1. Configuring the resource loader 55
11.2. Downloading files 55
11.3. Uploading files 56
11.4. Searching resources 57
11.5. Using CloudFormation 58
12. CloudWatch Metrics 60
13. Configuration properties 61
Spring Cloud Build 62
14. Building and Deploying 63
15. Contributing 64
15.1. Sign the Contributor License Agreement 64
15.2. Code of Conduct 64
15.3. Code Conventions and Housekeeping 64
15.4. Checkstyle 65
15.5. IDE setup 67
16. Flattening the POMs 71
17. Reusing the documentation 72
18. Updating the guides 75
Spring Cloud Bus 76
19. Quick Start 77
20. Bus Endpoints 78
20.1. Bus Refresh Endpoint 78
20.2. Bus Env Endpoint 78
21. Addressing an Instance 79
22. Addressing All Instances of a Service 80
23. Service ID Must Be Unique 81
24. Customizing the Message Broker 82
25. Tracing Bus Events 83
26. Broadcasting Your Own Events 84
26.1. Registering events in custom packages 84
27. Configuration properties 86

Spring Cloud Circuit Breaker . . . 87
.1. Configuring Resilience4J Circuit Breakers . . . 87
.2. Configuring Spring Retry Circuit Breakers . . . 89
28. Building . . . 92
28.1. Basic Compile and Test . . . 92
28.2. Documentation . . . 92
28.3. Working with the code . . . 93
29. Contributing . . . 94
29.1. Sign the Contributor License Agreement . . . 94
29.2. Code of Conduct . . . 94
29.3. Code Conventions and Housekeeping . . . 94
29.4. Checkstyle . . . 95
29.5. IDE setup . . . 97
Spring Boot Cloud CLI . . . 101
30. Installation . . . 102
31. Running Spring Cloud Services in Development . . . 103
31.1. Adding Additional Applications . . . 105
32. Writing Groovy Scripts and Running Applications . . . 106
33. Encryption and Decryption . . . 107
Spring Cloud for Cloud Foundry . . . 108
34. Discovery . . . 109
35. Single Sign On . . . 110
36. Configuration . . . 111
Cloud Native Applications . . . 112
37. Spring Cloud Context: Application Context Services . . . 113
37.1. The Bootstrap Application Context . . . 113
37.2. Application Context Hierarchies . . . 114
37.3. Changing the Location of Bootstrap Properties . . . 114
37.4. Overriding the Values of Remote Properties . . . 115
37.5. Customizing the Bootstrap Configuration . . . 115
37.6. Customizing the Bootstrap Property Sources . . . 116
37.7. Logging Configuration . . . 116
37.8. Environment Changes . . . 117
37.9. Refresh Scope . . . 117
37.10. Encryption and Decryption . . . 118
37.11. Endpoints . . . 118
38. Spring Cloud Commons: Common Abstractions . . . 120
38.1. The `@EnableDiscoveryClient` Annotation . . . 120
38.2. ServiceRegistry . . . 121
38.3. Spring RestTemplate as a Load Balancer Client . . . 122
38.4. Spring WebClient as a Load Balancer Client . . . 123

38.5. Multiple `RestTemplate` Objects . . . 126
38.6. Multiple WebClient Objects . . . 127
38.7. Spring WebFlux `WebClient` as a Load Balancer Client . . . 128
38.8. Ignore Network Interfaces . . . 130
38.9. HTTP Client Factories . . . 131
38.10. Enabled Features . . . 131
38.11. Spring Cloud Compatibility Verification . . . 132
39. Spring Cloud LoadBalancer . . . 134
39.1. Spring Cloud LoadBalancer integrations . . . 134
39.2. Spring Cloud LoadBalancer Caching . . . 134
39.3. Zone-Based Load-Balancing . . . 135
39.4. Instance Health-Check for LoadBalancer . . . 136
39.5. Spring Cloud LoadBalancer Starter . . . 137
39.6. Passing Your Own Spring Cloud LoadBalancer Configuration . . . 137
40. Spring Cloud Circuit Breaker . . . 140
40.1. Introduction . . . 140
40.2. Core Concepts . . . 140
40.3. Configuration . . . 141
41. CachedRandomPropertySource . . . 143
42. Configuration Properties . . . 144
Spring Cloud Config . . . 145
43. Quick Start . . . 146
43.1. Client Side Usage . . . 147
44. Spring Cloud Config Server . . . 151
44.1. Environment Repository . . . 152
44.2. Health Indicator . . . 176
44.3. Security . . . 176
44.4. Encryption and Decryption . . . 177
44.5. Key Management . . . 179
44.6. Creating a Key Store for Testing . . . 179
44.7. Using Multiple Keys and Key Rotation . . . 180
44.8. Serving Encrypted Properties . . . 180
45. Serving Alternative Formats . . . 181
46. Serving Plain Text . . . 182
46.1. Git, SVN, and Native Backends . . . 182
46.2. AWS S3 . . . 183
46.3. Decrypting Plain Text . . . 184
47. Embedding the Config Server . . . 185
48. Push Notifications and Spring Cloud Bus . . . 186
49. Spring Cloud Config Client . . . 187
49.1. Config First Bootstrap . . . 187

49.2. Discovery First Bootstrap 187
49.3. Config Client Fail Fast 188
49.4. Config Client Retry 188
49.5. Locating Remote Configuration Resources 188
49.6. Specifying Multiple Urls for the Config Server 189
49.7. Configuring Timeouts 189
49.8. Security 189
49.9. Nested Keys In Vault 192
Spring Cloud Consul 193
50. Install Consul 194
51. Consul Agent 195
52. Service Discovery with Consul 196
52.1. How to activate 196
52.2. Registering with Consul 196
52.3. HTTP Health Check 198
52.4. Looking up services 201
52.5. Consul Catalog Watch 202
53. Distributed Configuration with Consul 204
53.1. How to activate 204
53.2. Customizing 204
53.3. Config Watch 205
53.4. YAML or Properties with Config 205
53.5. git2consul with Config 206
53.6. Fail Fast 207
54. Consul Retry 208
55. Spring Cloud Bus with Consul 209
55.1. How to activate 209
56. Circuit Breaker with Hystrix 210
57. Hystrix metrics aggregation with Turbine and Consul 211
58. Configuration Properties 212
Spring Cloud Function 213
59. Introduction 214
60. Getting Started 216
61. Programming model 217
61.1. Function Catalog and Flexible Function Signatures 217
61.2. Java 8 function support 217
61.3. Function Composition 218
61.4. Function Routing 219
61.5. Function Arity 220
61.6. Type conversion (Content-Type negotiation) 220
61.7. Kotlin Lambda support 223

61.8. Function Component Scan 224
62. Standalone Web Applications 225
62.1. Function Mapping rules 226
62.2. Function Filtering rules 226
63. Standalone Streaming Applications 228
64. Deploying a Packaged Function 229
64.1. Supported Packaging Scenarios 230
65. Functional Bean Definitions 234
65.1. Comparing Functional with Traditional Bean Definitions 234
65.2. Limitations of Functional Bean Declaration 236
66. Testing Functional Applications 238
67. Dynamic Compilation 242
68. Serverless Platform Adapters 244
68.1. AWS Lambda 244
68.2. Microsoft Azure 251
68.3. Google Cloud Functions (Alpha) 254
Spring Cloud Gateway 260
69. How to Include Spring Cloud Gateway 261
70. Glossary 262
71. How It Works 263
72. Configuring Route Predicate Factories and Gateway Filter Factories 264
72.1. Shortcut Configuration 264
72.2. Fully Expanded Arguments 264
73. Route Predicate Factories 266
73.1. The After Route Predicate Factory 266
73.2. The Before Route Predicate Factory 266
73.3. The Between Route Predicate Factory 267
73.4. The Cookie Route Predicate Factory 267
73.5. The Header Route Predicate Factory 268
73.6. The Host Route Predicate Factory 268
73.7. The Method Route Predicate Factory 269
73.8. The Path Route Predicate Factory 269
73.9. The Query Route Predicate Factory 270
73.10. The RemoteAddr Route Predicate Factory 270
73.11. The Weight Route Predicate Factory 271
74. `GatewayFilter` Factories 273
74.1. The `AddRequestHeader` `GatewayFilter` Factory 273
74.2. The `AddRequestParameter` `GatewayFilter` Factory 274
74.3. The `AddResponseHeader` `GatewayFilter` Factory 274
74.4. The `DedupeResponseHeader` `GatewayFilter` Factory 275
74.5. The Hystrix `GatewayFilter` Factory 276

74.6. Spring Cloud CircuitBreaker GatewayFilter Factory ... 278
74.7. The FallbackHeaders GatewayFilter Factory ... 282
74.8. The MapRequestHeader GatewayFilter Factory ... 283
74.9. The PrefixPath GatewayFilter Factory ... 283
74.10. The PreserveHostHeader GatewayFilter Factory ... 284
74.11. The RequestRateLimiter GatewayFilter Factory ... 284
74.12. The RedirectTo GatewayFilter Factory ... 287
74.13. The RemoveRequestHeader GatewayFilter Factory ... 287
74.14. RemoveResponseHeader GatewayFilter Factory ... 288
74.15. The RemoveRequestParameter GatewayFilter Factory ... 288
74.16. The RewritePath GatewayFilter Factory ... 289
74.17. RewriteLocationResponseHeader GatewayFilter Factory ... 289
74.18. The RewriteResponseHeader GatewayFilter Factory ... 290
74.19. The SaveSession GatewayFilter Factory ... 291
74.20. The SecureHeaders GatewayFilter Factory ... 291
74.21. The SetPath GatewayFilter Factory ... 292
74.22. The SetRequestHeader GatewayFilter Factory ... 293
74.23. The SetResponseHeader GatewayFilter Factory ... 293
74.24. The SetStatus GatewayFilter Factory ... 294
74.25. The StripPrefix GatewayFilter Factory ... 295
74.26. The Retry GatewayFilter Factory ... 296
74.27. The RequestSize GatewayFilter Factory ... 297
74.28. The SetRequestHost GatewayFilter Factory ... 298
74.29. Modify a Request Body GatewayFilter Factory ... 299
74.30. Modify a Response Body GatewayFilter Factory ... 300
74.31. Default Filters ... 301
75. Global Filters ... 302
75.1. Combined Global Filter and GatewayFilter Ordering ... 302
75.2. Forward Routing Filter ... 303
75.3. The LoadBalancerClient Filter ... 303
75.4. The ReactiveLoadBalancerClientFilter ... 304
75.5. The Netty Routing Filter ... 304
75.6. The Netty Write Response Filter ... 305
75.7. The RouteToRequestUrl Filter ... 305
75.8. The Websocket Routing Filter ... 305
75.9. The Gateway Metrics Filter ... 306
75.10. Marking An Exchange As Routed ... 306
76. HttpHeadersFilters ... 308
76.1. Forwarded Headers Filter ... 308
76.2. RemoveHopByHop Headers Filter ... 308
76.3. XForwarded Headers Filter ... 308

77. TLS and SSL 310
77.1. TLS Handshake 311
78. Configuration 312
79. Route Metadata Configuration 313
80. Http timeouts configuration 314
80.1. Global timeouts 314
80.2. Per-route timeouts 314
80.3. Fluent Java Routes API 315
80.4. The `DiscoveryClient` Route Definition Locator 316
81. Reactor Netty Access Logs 318
82. CORS Configuration 319
83. Actuator API 320
83.1. Verbose Actuator Format 320
83.2. Retrieving Route Filters 321
83.3. Refreshing the Route Cache 322
83.4. Retrieving the Routes Defined in the Gateway 322
83.5. Retrieving Information about a Particular Route 323
83.6. Creating and Deleting a Particular Route 324
83.7. Recap: The List of All endpoints 324
84. Troubleshooting 326
84.1. Log Levels 326
84.2. Wiretap 326
85. Developer Guide 327
85.1. Writing Custom Route Predicate Factories 327
85.2. Writing Custom GatewayFilter Factories 327
85.3. Writing Custom Global Filters 329
86. Building a Simple Gateway by Using Spring MVC or Webflux 331
87. Configuration properties 333
Spring Cloud GCP 334
88. Introduction 335
89. Getting Started 336
89.1. Setting up Dependencies 336
89.2. Learning Spring Cloud GCP 338
90. Spring Cloud GCP Core 340
90.1. Configuration 340
90.2. Project ID 340
90.3. Credentials 341
90.4. Environment 342
90.5. Spring Initializr 343
91. Cloud Storage 344
91.1. Using Cloud Storage 344

91.2. Cloud Storage Objects As Spring Resources 344
91.3. Configuration 345
91.4. Sample 346
92. Cloud SQL 347
92.1. Prerequisites 347
92.2. Spring Boot Starter for Google Cloud SQL 347
92.3. Samples 350
93. Cloud Pub/Sub 351
93.1. Configuration 351
93.2. Spring Boot Actuator Support 355
93.3. Pub/Sub Operations & Template 356
93.4. Reactive Stream Subscription 361
93.5. Pub/Sub management 361
93.6. Sample 364
94. Spring Integration 365
94.1. Channel Adapters for Cloud Pub/Sub 365
94.2. Channel Adapters for Google Cloud Storage 372
95. Spring Cloud Stream 375
95.1. Overview 375
95.2. Configuration 375
95.3. Binding with Functions 377
95.4. Binding with Annotations 377
95.5. Streaming vs. Polled Input 378
95.6. Sample 379
96. Spring Cloud Bus 380
96.1. Configuration Management with Spring Cloud Config and Spring Cloud Bus 380
97. Stackdriver Trace 382
97.1. Tracing 382
97.2. Spring Boot Starter for Stackdriver Trace 383
97.3. Overriding the auto-configuration 385
97.4. Customizing spans 385
97.5. Integration with Logging 386
97.6. Sample 386
98. Stackdriver Logging 387
98.1. Web MVC Interceptor 387
98.2. Logback Support 388
98.3. Sample 391
99. Stackdriver Monitoring 392
99.1. Spring Boot Starter for Stackdriver Monitoring 392
100. Spring Data Cloud Spanner 394
100.1. Configuration 394

100.2. Object Mapping 397
100.3. Spanner Operations & Template 408
100.4. Repositories 414
100.5. Query Methods 416
100.6. Database and Schema Admin 422
100.7. Events 423
100.8. Auditing 424
100.9. Multi-Instance Usage 425
100.10. Cloud Spanner Emulator 426
100.11. Sample 426
101. Spring Data Cloud Datastore 427
101.1. Configuration 428
101.2. Object Mapping 430
101.3. Relationships 437
101.4. Datastore Operations & Template 443
101.5. Repositories 446
101.6. Events 455
101.7. Auditing 455
101.8. Partitioning Data by Namespace 457
101.9. Spring Boot Actuator Support 457
101.10. Sample 457
102. Spring Data Cloud Firestore 458
102.1. Configuration 458
102.2. Object Mapping 460
102.3. Reactive Repositories 463
102.4. Firestore Operations & Template 464
102.5. Query methods by convention 465
102.6. Transactions 467
102.7. Cloud Firestore Spring Boot Starter 470
102.8. Emulator Usage 471
102.9. Samples 472
103. Cloud Memorystore for Redis 473
103.1. Spring Caching 473
104. BigQuery 474
104.1. Configuration 474
104.2. Spring Integration 476
104.3. Sample 477
105. Cloud IAP 478
105.1. Configuration 479
105.2. Sample 479
106. Cloud Vision 480

106.1. Dependency Setup . . . 480
106.2. Configuration . . . 480
106.3. Image Analysis . . . 481
106.4. Document OCR Template . . . 482
106.5. Sample . . . 484
107. Secret Manager . . . 485
107.1. Dependency Setup . . . 485
107.2. Secret Manager Property Source . . . 486
107.3. Secret Manager Template . . . 487
107.4. Sample . . . 487
108. Cloud Runtime Configuration API . . . 488
108.1. Configuration . . . 488
108.2. Quick start . . . 489
108.3. Refreshing the configuration at runtime . . . 490
108.4. Sample . . . 491
109. Cloud Foundry . . . 492
109.1. User-Provided Services . . . 492
110. Kotlin Support . . . 494
110.1. Prerequisites . . . 494
110.2. Sample . . . 494
111. Configuration properties . . . 495
Spring Cloud Kubernetes . . . 496
112. Why do you need Spring Cloud Kubernetes? . . . 497
113. Starters . . . 498
114. DiscoveryClient for Kubernetes . . . 499
115. Kubernetes native service discovery . . . 501
116. Kubernetes PropertySource implementations . . . 502
116.1. Using a `ConfigMap` `PropertySource` . . . 502
116.2. Secrets PropertySource . . . 508
116.3. `PropertySource` Reload . . . 512
117. Ribbon Discovery in Kubernetes . . . 515
118. Kubernetes Ecosystem Awareness . . . 517
118.1. Kubernetes Profile Autoconfiguration . . . 517
118.2. Istio Awareness . . . 517
119. Pod Health Indicator . . . 518
120. Leader Election . . . 519
121. LoadBalancer for Kubernetes . . . 520
122. Security Configurations Inside Kubernetes . . . 521
122.1. Namespace . . . 521
122.2. Service Account . . . 521
123. Service Registry Implementation . . . 523

124. Examples 524
125. Other Resources 525
126. Configuration properties 526
127. Building 527
127.1. Basic Compile and Test 527
127.2. Documentation 527
127.3. Working with the code 528
128. Contributing 529
128.1. Sign the Contributor License Agreement 529
128.2. Code of Conduct 529
128.3. Code Conventions and Housekeeping 529
128.4. Checkstyle 530
128.5. IDE setup 532
Spring Cloud Netflix 536
129. Service Discovery: Eureka Clients 537
129.1. How to Include Eureka Client 537
129.2. Registering with Eureka 537
129.3. Authenticating with the Eureka Server 538
129.4. Status Page and Health Indicator 539
129.5. Registering a Secure Application 539
129.6. Eureka's Health Checks 540
129.7. Eureka Metadata for Instances and Clients 541
129.8. Using the EurekaClient 542
129.9. Alternatives to the Native Netflix EurekaClient 543
129.10. Why Is It so Slow to Register a Service? 544
129.11. Zones 544
129.12. Refreshing Eureka Clients 545
129.13. Using Eureka with Spring Cloud LoadBalancer 545
130. Service Discovery: Eureka Server 546
130.1. How to Include Eureka Server 546
130.2. How to Run a Eureka Server 546
130.3. High Availability, Zones and Regions 547
130.4. Standalone Mode 547
130.5. Peer Awareness 548
130.6. When to Prefer IP Address 549
130.7. Securing The Eureka Server 550
130.8. Disabling Ribbon with Eureka Server and Client starters 550
130.9. JDK 11 Support 551
131. Circuit Breaker: Spring Cloud Circuit Breaker With Hystrix 552
131.1. Disabling Spring Cloud Circuit Breaker Hystrix 552
131.2. Configuring Hystrix Circuit Breakers 552

132. Circuit Breaker: Hystrix Clients . . . 554
132.1. How to Include Hystrix . . . 555
132.2. Propagating the Security Context or Using Spring Scopes . . . 556
132.3. Health Indicator . . . 557
132.4. Hystrix Metrics Stream . . . 557
133. Circuit Breaker: Hystrix Dashboard . . . 558
134. Hystrix Timeouts And Ribbon Clients . . . 559
134.1. How to Include the Hystrix Dashboard . . . 559
134.2. Turbine . . . 559
134.3. Turbine Stream . . . 561
135. Client Side Load Balancer: Ribbon . . . 563
135.1. How to Include Ribbon . . . 563
135.2. Customizing the Ribbon Client . . . 563
135.3. Customizing the Default for All Ribbon Clients . . . 564
135.4. Customizing the Ribbon Client by Setting Properties . . . 565
135.5. Using Ribbon with Eureka . . . 566
135.6. Example: How to Use Ribbon Without Eureka . . . 567
135.7. Example: Disable Eureka Use in Ribbon . . . 567
135.8. Using the Ribbon API Directly . . . 567
135.9. Caching of Ribbon Configuration . . . 568
135.10. How to Configure Hystrix Thread Pools . . . 568
135.11. How to Provide a Key to Ribbon’s `IRule` . . . 569
136. External Configuration: Archaius . . . 570
137. Router and Filter: Zuul . . . 571
137.1. How to Include Zuul . . . 571
137.2. Embedded Zuul Reverse Proxy . . . 571
137.3. Zuul Http Client . . . 576
137.4. Cookies and Sensitive Headers . . . 576
137.5. Ignored Headers . . . 577
137.6. Management Endpoints . . . 577
137.7. Strangulation Patterns and Local Forwards . . . 578
137.8. Uploading Files through Zuul . . . 579
137.9. Query String Encoding . . . 580
137.10. Request URI Encoding . . . 580
137.11. Plain Embedded Zuul . . . 581
137.12. Disable Zuul Filters . . . 581
137.13. Providing Hystrix Fallbacks For Routes . . . 581
137.14. Zuul Timeouts . . . 583
137.15. Rewriting the `Location` header . . . 584
137.16. Enabling Cross Origin Requests . . . 584
137.17. Metrics . . . 585

137.18. Zuul Developer Guide 585
138. Polyglot support with Sidecar 591
139. Retrying Failed Requests 593
139.1. BackOff Policies 593
139.2. Configuration 593
140. HTTP Clients 595
141. Modules In Maintenance Mode 596
142. Configuration properties 597
Spring Cloud OpenFeign 598
143. Declarative REST Client: Feign 599
143.1. How to Include Feign 599
143.2. Overriding Feign Defaults 600
143.3. Creating Feign Clients Manually 604
143.4. Feign Hystrix Support 605
143.5. Feign Hystrix Fallbacks 606
143.6. Feign and `@Primary` 607
143.7. Feign Inheritance Support 607
143.8. Feign request/response compression 608
143.9. Feign logging 609
143.10. Feign @QueryMap support 609
143.11. HATEOAS support 610
143.12. Spring @MatrixVariable Support 610
143.13. Feign `CollectionFormat` support 611
143.14. Reactive Support 612
144. Configuration properties 613
Spring Cloud Security 614
145. Quickstart 615
145.1. OAuth2 Single Sign On 615
145.2. OAuth2 Protected Resource 616
146. More Detail 618
146.1. Single Sign On 618
146.2. Token Relay 618
147. Configuring Authentication Downstream of a Zuul Proxy 622
Spring Cloud Sleuth 623
148. Introduction 624
148.1. Terminology 624
148.2. Purpose 625
148.3. Adding Sleuth to the Project 634
148.4. Overriding the auto-configuration of Zipkin 638
149. Additional Resources 640
150. Features 641

150.1. Introduction to Brave . . . 642
151. Sampling . . . 648
151.1. Declarative sampling . . . 648
151.2. Custom sampling . . . 648
151.3. Sampling in Spring Cloud Sleuth . . . 649
152. Propagation . . . 650
152.1. Propagating extra fields . . . 651
153. Current Tracing Component . . . 655
154. Current Span . . . 656
154.1. Setting a span in scope manually . . . 656
155. Instrumentation . . . 657
156. Span lifecycle . . . 658
156.1. Creating and finishing spans . . . 658
156.2. Continuing Spans . . . 659
156.3. Creating a Span with an explicit Parent . . . 659
157. Naming spans . . . 661
157.1. `@SpanName` Annotation . . . 661
157.2. `toString()` method . . . 661
158. Managing Spans with Annotations . . . 663
158.1. Rationale . . . 663
158.2. Creating New Spans . . . 663
158.3. Continuing Spans . . . 664
158.4. Advanced Tag Setting . . . 664
159. Customizations . . . 667
159.1. Customizers . . . 667
159.2. HTTP . . . 667
159.3. `TracingFilter` . . . 669
159.4. Messaging . . . 669
159.5. RPC . . . 670
159.6. Custom service name . . . 670
159.7. Customization of Reported Spans . . . 671
159.8. Host Locator . . . 671
160. Sending Spans to Zipkin . . . 673
161. Zipkin Stream Span Consumer . . . 675
162. Integrations . . . 676
162.1. OpenTracing . . . 676
162.2. Runnable and Callable . . . 676
162.3. Spring Cloud CircuitBreaker . . . 677
162.4. Hystrix . . . 677
162.5. RxJava . . . 678
162.6. HTTP integration . . . 678

162.7. HTTP Client Integration . . . 680
162.8. Feign . . . 682
162.9. gRPC . . . 682
162.10. Asynchronous Communication . . . 684
162.11. Messaging . . . 686
162.12. Zuul . . . 687
162.13. Redis . . . 687
162.14. Quartz . . . 687
162.15. Project Reactor . . . 687
163. Configuration properties . . . 688
164. Running examples . . . 689
Spring Cloud Task Reference Guide . . . 690
Preface . . . 691
165. About the documentation . . . 692
166. Getting help . . . 693
167. First Steps . . . 694
Getting started . . . 695
168. Introducing Spring Cloud Task . . . 696
169. System Requirements . . . 697
169.1. Database Requirements . . . 697
170. Developing Your First Spring Cloud Task Application . . . 698
170.1. Creating the Spring Task Project using Spring Initializr . . . 698
170.2. Writing the Code . . . 698
170.3. Running the Example . . . 700
Features . . . 703
171. The lifecycle of a Spring Cloud Task . . . 704
171.1. The TaskExecution . . . 705
171.2. Mapping Exit Codes . . . 705
172. Configuration . . . 707
172.1. DataSource . . . 707
172.2. Table Prefix . . . 707
172.3. Enable/Disable table initialization . . . 707
172.4. Externally Generated Task ID . . . 707
172.5. External Task Id . . . 708
172.6. Parent Task Id . . . 708
172.7. TaskConfigurer . . . 708
172.8. Task Name . . . 709
172.9. Task Execution Listener . . . 709
172.10. Restricting Spring Cloud Task Instances . . . 711
172.11. Disabling Spring Cloud Task Auto Configuration . . . 712
172.12. Closing the Context . . . 712

Batch ... 713
173. Associating a Job Execution to the Task in which It Was Executed ... 714
173.1. Overriding the TaskBatchExecutionListener ... 714
174. Remote Partitioning ... 715
174.1. Notes on Developing a Batch-partitioned application for the Kubernetes Platform ... 716
174.2. Notes on Developing a Batch-partitioned Application for the Cloud Foundry Platform 717
175. Batch Informational Messages ... 719
176. Batch Job Exit Codes ... 720
Spring Cloud Stream Integration ... 721
177. Launching a Task from a Spring Cloud Stream ... 722
177.1. Spring Cloud Data Flow ... 723
178. Spring Cloud Task Events ... 724
178.1. Disabling Specific Task Events ... 724
179. Spring Batch Events ... 725
179.1. Sending Batch Events to Different Channels ... 725
179.2. Disabling Batch Events ... 725
179.3. Emit Order for Batch Events ... 726
Appendices ... 727
180. Task Repository Schema ... 728
180.1. Table Information ... 728
181. Building This Documentation ... 731
182. Running a Task App on Cloud Foundry ... 732
Spring Cloud Vault ... 733
183. Quick Start ... 734
184. Client Side Usage ... 737
184.1. Authentication ... 739
185. Authentication methods ... 740
185.1. Token authentication ... 740
185.2. Vault Agent authentication ... 740
185.3. AppId authentication ... 741
185.4. AppRole authentication ... 742
185.5. AWS-EC2 authentication ... 744
185.6. AWS-IAM authentication ... 746
185.7. Azure MSI authentication ... 747
185.8. TLS certificate authentication ... 747
185.9. Cubbyhole authentication ... 748
185.10. GCP-GCE authentication ... 749
185.11. GCP-IAM authentication ... 750
185.12. Kubernetes authentication ... 751
185.13. Pivotal CloudFoundry authentication ... 752
186. Secret Backends ... 753

186.1. Generic Backend 753
186.2. Key-Value Backend 754
186.3. Consul 755
186.4. RabbitMQ 756
186.5. AWS 757
187. Database backends 759
187.1. Database 759
187.2. Apache Cassandra 760
187.3. MongoDB 761
187.4. MySQL 762
187.5. PostgreSQL 762
188. Configure `PropertySourceLocator` behavior 764
189. Service Registry Configuration 765
190. Vault Client Fail Fast 766
191. Vault Enterprise Namespace Support 767
192. Vault Client SSL configuration 768
193. Lease lifecycle management (renewal and revocation) 769
Spring Cloud Zookeeper 770
194. Install Zookeeper 771
195. Service Discovery with Zookeeper 772
195.1. Activating 772
195.2. Registering with Zookeeper 772
195.3. Using the DiscoveryClient 773
196. Using Spring Cloud Zookeeper with Spring Cloud Netflix Components 774
196.1. Ribbon with Zookeeper 774
197. Spring Cloud Zookeeper and Service Registry 775
197.1. Instance Status 775
198. Zookeeper Dependencies 776
198.1. Using the Zookeeper Dependencies 776
198.2. Activating Zookeeper Dependencies 776
198.3. Setting up Zookeeper Dependencies 776
198.4. Configuring Spring Cloud Zookeeper Dependencies 779
199. Spring Cloud Zookeeper Dependency Watcher 781
199.1. Activating 781
199.2. Registering a Listener 781
199.3. Using the Presence Checker 781
200. Distributed Configuration with Zookeeper 782
200.1. Activating 782
200.2. Customizing 782
200.3. Access Control Lists (ACLs) 783
Appendix: Compendium of Configuration Properties 785

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

Release Train Version: **Hoxton.SR8**

Supported Boot Version: **2.3.3.RELEASE**

Chapter 1. Features

Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Distributed messaging

Chapter 2. Release Train Versions

Table 1. Release Train Project Versions

Project Name	Project Version
spring-boot	2.3.3.RELEASE
spring-cloud-aws	2.2.4.RELEASE
spring-cloud-build	2.3.1.RELEASE
spring-cloud-bus	2.2.3.RELEASE
spring-cloud-circuitbreaker	1.0.4.RELEASE
spring-cloud-cli	2.2.2.RELEASE
spring-cloud-cloudfoundry	2.2.3.RELEASE
spring-cloud-commons	2.2.5.RELEASE
spring-cloud-config	2.2.5.RELEASE
spring-cloud-consul	2.2.4.RELEASE
spring-cloud-contract	2.2.4.RELEASE
spring-cloud-function	3.0.10.RELEASE
spring-cloud-gateway	2.2.5.RELEASE
spring-cloud-gcp	1.2.5.RELEASE
spring-cloud-kubernetes	1.1.6.RELEASE
spring-cloud-netflix	2.2.5.RELEASE
spring-cloud-openfeign	2.2.5.RELEASE
spring-cloud-security	2.2.4.RELEASE
spring-cloud-sleuth	2.2.5.RELEASE
spring-cloud-task	2.2.3.RELEASE
spring-cloud-vault	2.2.5.RELEASE
spring-cloud-zookeeper	2.2.3.RELEASE

Spring Cloud AWS

Spring Cloud for Amazon Web Services, part of the Spring Cloud umbrella project, eases the integration with hosted Amazon Web Services. It offers a convenient way to interact with AWS provided services using well-known Spring idioms and APIs, such as the messaging or caching API. Developers can build their application around the hosted services without having to care about infrastructure or maintenance.

Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at github.

Chapter 3. Using Amazon Web Services

Amazon provides a Java SDK to issue requests for the all services provided by the Amazon Web Service platform. Using the SDK, application developers still have to integrate the SDK into their application with a considerable amount of infrastructure related code. Spring Cloud AWS provides application developers already integrated Spring-based modules to consume services and avoid infrastructure related code as much as possible. The Spring Cloud AWS module provides a module set so that application developers can arrange the dependencies based on their needs for the particular services. The graphic below provides a general overview of all Spring Cloud AWS modules along with the service support for the respective Spring Cloud AWS services.

[Overview] | *overview.png*

- **Spring Cloud AWS Core** is the core module of Spring Cloud AWS providing basic services for security and configuration setup. Developers will not use this module directly but rather through other modules. The core module provides support for cloud based environment configurations providing direct access to the instance based EC2 metadata and the overall application stack specific CloudFormation metadata.
- **Spring Cloud AWS Context** delivers access to the Simple Storage Service via the Spring resource loader abstraction. Moreover developers can send e-mails using the Simple E-Mail Service and the Spring mail abstraction. Further the developers can introduce declarative caching using the Spring caching support and the ElastiCache caching service.
- **Spring Cloud AWS JDBC** provides automatic datasource lookup and configuration for the Relational Database Service which can be used with JDBC or any other support data access technology by Spring.
- **Spring Cloud AWS Messaging** enables developers to receive and send messages with the Simple Queueing Service for point-to-point communication. Publish-subscribe messaging is supported with the integration of the Simple Notification Service.
- **Spring Cloud AWS Parameter Store Configuration** enables Spring Cloud applications to use the AWS Parameter Store as a Bootstrap Property Source, comparable to the support provided for the Spring Cloud Config Server or Consul's key-value store.
- **Spring Cloud AWS Secrets Manager Configuration** enables Spring Cloud applications to use the AWS Secrets Manager as a Bootstrap Property Source, comparable to the support provided for the Spring Cloud Config Server or Consul's key-value store.

Chapter 4. Basic setup

Before using the Spring Cloud AWS module developers have to pick the dependencies and configure the Spring Cloud AWS module. The next chapters describe the dependency management and also the basic configuration for the Spring AWS Cloud project.

4.1. Spring Cloud AWS maven dependency management

Spring Cloud AWS module dependencies can be used directly in Maven with a direct configuration of the particular module. The Spring Cloud AWS module includes all transitive dependencies for the Spring modules and also the Amazon SDK that are needed to operate the modules. The general dependency configuration will look like this:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-aws-context</artifactId>
    <version>{spring-cloud-version}</version>
  </dependency>
</dependencies>
```

Different modules can be included by replacing the module name with the respective one (e.g. `spring-cloud-aws-messaging` instead of `spring-cloud-aws-context`)

The example above works with the Maven Central repository. To use the Spring Maven repository (e.g. for milestones or developer snapshots), you need to specify the repository location in your Maven configuration. For full releases:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.release</id>
        <url>https://repo.spring.io/release/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

For milestones:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.milestone</id>
        <url>https://repo.spring.io/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

4.2. Amazon SDK dependency version management

Amazon SDK is released more frequently than Spring Cloud AWS. If you need to use newer version of AWS SDK than one configured by Spring Cloud AWS add AWS SDK BOM to dependency management section making sure it is declared before any other BOM dependency that configures AWS SDK dependencies.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-java-sdk-bom</artifactId>
            <version>${aws-java-sdk.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

4.3. Amazon SDK configuration

The Spring Cloud AWS configuration is currently done using custom elements provided by Spring Cloud AWS namespaces. JavaConfig will be supported soon. The configuration setup is done directly in Spring XML configuration files so that the elements can be directly used. Each module of Spring Cloud AWS provides custom namespaces to allow the modular use of the modules. A typical XML configuration to use Spring Cloud AWS is outlined below:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aws-context="http://www.springframework.org/schema/cloud/aws/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cloud/aws/context
        http://www.springframework.org/schema/cloud/aws/context/spring-cloud-aws-
context.xsd">

           <aws-context:context-region region="..."/>
</beans>
```

On application startup, for its internal purposes Spring Cloud AWS performs a check if application runs in AWS cloud environment by using `EC2MetadataUtils` class provided by AWS SDK. Starting from version 1.11.678, AWS SDK logs a warning message with exception when this check is made outside of AWS environment. This warning message can be hidden by setting `ERROR` logging level on `com.amazonaws.util.EC2MetadataUtils` class.

```
logging.level.com.amazonaws.util.EC2MetadataUtils=error
```

4.3.1. SDK credentials configuration

In order to make calls to the Amazon Web Service the credentials must be configured for the the Amazon SDK. Spring Cloud AWS provides support to configure an application context specific credentials that are used for *each* service call for requests done by Spring Cloud AWS components, with the exception of the Parameter Store and Secrets Manager Configuration. Therefore there must be **exactly one** configuration of the credentials for an entire application context.

The `com.amazonaws.auth.DefaultAWSCredentialsProviderChain` is used by all the clients if there is no dedicated credentials provider defined. This will essentially use the following authentication information

- use the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`
- use the system properties `aws.accessKeyId` and `aws.secretKey`
- use the user specific profile credentials file
- use ECS credentials if the `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` environment variable is set
- use the instance profile credentials (see below)

Based on the overall credentials policy there are different options to configure the credentials. The possible ones are described in the following sub-chapters.

Simple credentials configuration

Credentials for the Amazon SDK consist of an access key (which might be shared) and a secret key (which must **not** be shared). Both security attributes can be configured using the XML namespaces for each Amazon SDK service created by the Spring Cloud AWS module. The overall configuration looks like this

```
<beans ...>
  <aws-context:context-credentials>
   <aws-context:simple-credentials access-key="AKIAIO" secret-key="wJalrXUtnFEMI/K7M"
/>
  </aws-context:context-credentials>
</beans>
```

The access-key and secret-key should be externalized into property files (e.g. Spring Boot application configuration) and not be checked in into the source management system.

Instance profile configuration

An instance profile configuration allows to assign a profile that is authorized by a role while starting an EC2 instance. All calls made from the EC2 instance are then authenticated with the instance profile specific user role. Therefore there is no dedicated access-key and secret-key needed in the configuration. The configuration for the instance profile in Spring Cloud AWS looks like this:

```
<beans ...>
    <aws-context:context-credentials>
        <aws-context:instance-profile-credentials/>
    </aws-context:context-credentials>
</beans>
```

Mixing both security configurations

In some cases it is useful to combine both authentication strategies to allow the application to use the instance profile with a fallback for an explicit access-key and secret-key configuration. This is useful if the application is tested inside EC2 (e.g. on a test server) and locally for testing. The next snippet shows a combination of both security configurations.

```
<beans ...>
   <aws-context:context-credentials>
       <aws-context:instance-profile-credentials/>
       <aws-context:simple-credentials access-key="${accessKey:}" secret-
key="${secretKey:}"/>
   </aws-context:context-credentials>
</beans>
```

The access-key and secret-key are defined using a placeholder expressions along with a default value to avoid bootstrap errors if the properties are not configured at all.

Parameter Store and Secrets Manager Configuration credentials and region configuration

The Parameter Store and Secrets Manager Configuration support uses a bootstrap context to configure a default `AWSSimpleSystemsManagement` client, which uses a `com.amazonaws.auth.DefaultAWSCredentialsProviderChain` and `com.amazonaws.regions.DefaultAwsRegionProviderChain`. If you want to override this, then you need to define your own Spring Cloud bootstrap configuration class with a bean of type `AWSSimpleSystemsManagement` that's configured to use your chosen credentials and/or region provider. Because this context is created when your Spring Cloud Bootstrap context is created, you can't simply override the bean in a regular `@Configuration` class.

4.3.2. Region configuration

Amazon Web services are available in different regions. Based on the custom requirements, the user can host the application on different Amazon regions. The `spring-cloud-aws-context` module provides a way to define the region for the entire application context.

Explicit region configuration

The region can be explicitly configured using an XML element. This is particularly useful if the region can not be automatically derived because the application is not hosted on a EC2 instance (e.g. local testing) or the region must be manually overridden.

```
<beans ...>
    <aws-context:context-region region="eu-west-1"/>
</beans>
```

It is also allowed to use expressions or placeholders to externalize the configuration and ensure that the region can be reconfigured with property files or system properties.

Automatic region configuration

If the application context is started inside an EC2 instance, then the region can automatically be fetched from the instance metadata and therefore must not be configured statically. The configuration will look like this:

```
<beans ...>
  <aws-context:context-region auto-detect="true" />
</beans>
```

Service specific region configuration

A region can also be overridden for particular services if one application context consumes services from different regions. The configuration can be done globally like described above and configured for each service with a region attribute. The configuration might look like this for a database service (described later)

```
<beans ...>
 <aws-context:context-region region="eu-central-1" />
 <jdbc:data-source ... region="eu-west-1" />
</beans>
```

While it is theoretically possible to use multiple regions per application, we strongly recommend to write applications that are hosted only inside one region and split the application if it is hosted in different regions at the same time.

4.3.3. Spring Boot auto-configuration

Following the Spring Cloud umbrella project, Spring Cloud AWS also provides dedicated Spring Boot support. Spring Cloud AWS can be configured using Spring Boot properties and will also automatically guess any sensible configuration based on the general setup.

Maven dependencies

Spring Cloud AWS provides a dedicated module to enable the Spring Boot support. That module must be added to the general maven dependency inside the application. The typical configuration will look like this

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-aws-autoconfigure</artifactId>
    <version>{spring-cloud-version}</version>
  </dependency>
</dependencies>
```

Additional dependencies to enable particular features like messaging and JDBC have to be added. Spring Cloud AWS will only configure classes that are available in the Spring Boot application's classpath.

Configuring credentials

Spring Boot provides a standard way to define properties with property file or YAML configuration files. Spring Cloud AWS provides support to configure the credential information with the Spring Boot application configuration files. Spring Cloud AWS provides the following properties to configure the credentials setup for the whole application.

Unless `cloud.aws.credentials.use-default-aws-credentials-chain` is set to `true`, Spring Cloud AWS configures following credentials chain:

1. `AWSStaticCredentialsProvider` if `cloud.aws.credentials.access-key` is provided
2. `EC2ContainerCredentialsProviderWrapper` unless `cloud.aws.credentials.instance-profile` is set to `false`
3. `ProfileCredentialsProvider`

property	example	description
cloud.aws.credentials.access-key	AKIAIOSFODNN7EXAMPLE	The access key to be used with a static provider
cloud.aws.credentials.secret-key	wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY	The secret key to be used with a static provider
cloud.aws.credentials.instance-profile	true	Configures an instance profile credentials provider with no further configuration
cloud.aws.credentials.profile-name	default	The name of a configuration profile in the specified configuration file
cloud.aws.credentials.profile-path	`~/.aws/credentials`	The file path where the profile configuration file is located. Defaults to `~/.aws/credentials` if value is not provided
cloud.aws.credentials.use-default-aws-credentials-chain	true	Use the DefaultAWSCredentials Chain instead of configuring a custom credentials chain

Configuring region

Like for the credentials, the Spring Cloud AWS module also supports the configuration of the region inside the Spring Boot configuration files. The region can be automatically detected or explicitly configured (e.g. in case of local tests against the AWS cloud).

The properties to configure the region are shown below

property	example	description
cloud.aws.region.auto	true	Enables automatic region detection based on the EC2 meta data service
cloud.aws.region.use-default-aws-region-chain	true	Use the DefaultAWSRegion Chain instead of configuring a custom region chain

property	example	description
cloud.aws.region.static	eu-west-1	Configures a static region for the application. Possible regions are (currently) us-east-1, us-west-1, us-west-2, eu-west-1, eu-central-1, ap-southeast-1, ap-southeast-1, ap-northeast-1, sa-east-1, cn-north-1 and any custom region configured with own region meta data

Chapter 5. Cloud environment

Applications often need environment specific configuration information, especially in changing environments like in the Amazon cloud environment. Spring Cloud AWS provides a support to retrieve and use environment specific data inside the application context using common Spring mechanisms like property placeholder or the Spring expression language.

5.1. Retrieving instance metadata

Instance metadata are available inside an EC2 environment. The metadata can be queried using a special HTTP address that provides the instance metadata. Spring Cloud AWS enables application to access this metadata directly in expression or property placeholder without the need to call an external HTTP service.

5.1.1. Enabling instance metadata support with XML

The instance metadata retrieval support is enabled through an XML element like the standard property placeholder in Spring. The following code sample demonstrates the activation of the instance metadata support inside an application context.

```
<beans ...>
     <aws-context:context-instance-data />
</beans>
```

Instance metadata can be retrieved without an authorized service call, therefore the configuration above does not require any region or security specific configuration.

5.1.2. Enabling instance metadata support with Java

The instance metadata can also be configured within a Java configuration class without the need for an XML configuration. The next example shows a typical Spring `@Configuration` class that enables the instance metadata with the `org.springframework.cloud.aws.context.config.annotation.EnableInstanceData`

```
@Configuration
@EnableContextInstanceData
public static class ApplicationConfiguration {
}
```

5.1.3. Enabling instance metadata support in Spring Boot

The instance metadata is automatically available in a Spring Boot application as a property source if the application is running on an EC2 instance.

5.1.4. Using instance metadata

Instance metadata can be used in XML, Java placeholders and expressions. The example below demonstrates the usage of instance metadata inside an XML file using placeholders and also the expression referring to the special variable `environment`

```
<beans ...>
 <bean class="org.springframework.cloud.aws....SimpleConfigurationBean">
    <property name="value1" value="#{environment.ami-id}" />
    <property name="value2" value="#{environment.hostname}" />
    <property name="value3" value="${instance-type}" />
    <property name="value4" value="${instance-id}" />
 </bean>
</beans>
```

Instance metadata can also be injected with the Spring `org.springframework.beans.factory.annotation.Value` annotation directly into Java fields. The next example demonstrates the use of instance metadata inside a Spring bean.

```
@Component
public class ApplicationInfoBean {

    @Value("${ami-id:N/A}")
    private String amiId;

    @Value("${hostname:N/A}")
    private String hostname;

    @Value("${instance-type:N/A}")
    private String instanceType;

    @Value("${services/domain:N/A}")
    private String serviceDomain;
}
```

Every instance metadata can be accessed by the key available in the instance metadata service Nested properties can be accessed by separating the properties with a slash ('/').

5.1.5. Using instance user data

Besides the default instance metadata it is also possible to configure user data on each instance. This user data is retrieved and parsed by Spring Cloud AWS. The user data can be defined while starting an EC2 instance with the application. Spring Cloud AWS expects the format `<key>:<value>;<key>:<value>` inside the user data so that it can parse the string and extract the key value pairs.

The user data can be configured using either the management console shown below or a CloudFormation template.

[User data in the management console] | *cloud-environment-user-data.png*

A CloudFormation template snippet for the configuration of the user data is outlined below:

```
...
"Resources": {
 "ApplicationServerInstance": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
   "ImageId": "ami-6a56b81d",
   "UserData": {
      "Fn::Base64": "data1:value1;data2:value2"
    },
   "InstanceType": "t1.micro",
  }
}
...
```

The user data can be accessed directly in the application context like the instance metadata through placeholders or expressions.

```
@Component
public class SecondConfigurationBean {

    @Value("${data1}")
    private String firstDataOption;

    @Value("${data2}")
    private String secondDataOption;
}
```

5.1.6. Using instance tags

User configured properties can also be configured with tags instead of user data. Tags are a global concept in the context of Amazon Web services and used in different services. Spring Cloud AWS supports instance tags also across different services. Compared to user data, user tags can be updated during runtime, there is no need to stop and restart the instance.

User data can also be used to execute scripts on instance startup. Therefore it is useful to leverage instance tags for user configuration and user data to execute scripts on instance startup.

Instance specific tags can be configured on the instance level through the management console outlined below and like user data also with a CloudFormation template shown afterwards.

[Instance data in the management console] | *cloud-environment-instance-tags.png*

A CloudFormation template snippet for the configuration of the instance tags is outlined below:

```
...
"Resources": {
    "UserTagAndUserDataInstance": {
        "Type": "AWS::EC2::Instance",
        "Properties": {
            "ImageId": "ami-6a56b81d",
            "InstanceType": "t1.micro",
            "Tags": [
            {
                "Key": "tag1",
                "Value": "tagv1"
            },
            {
                "Key": "tag3",
                "Value": "tagv3"
            },
            {
                "Key": "tag2",
                "Value": "tagv2"
            },
            {
                "Key": "tag4",
                "Value": "tagv4"
            }
            ]
        }
    }
}
...
```

To retrieve the instance tags, Spring Cloud AWS has to make authenticated requests and therefore it will need the region and security configuration before actually resolving the placeholders. Also because the instance tags are not available while starting the application context, they can only be referenced as expressions and not with placeholders. The `context-instance-data` element defines an attribute `user-tags-map` that will create a map in the application context for the name. This map can then be queried using expression for other bean definitions.

```
<beans ...>
 <aws-context:context-instance-data user-tags-map="instanceData" />
</beans>
```

A java bean might resolve expressions with the `@Value` annotation.

```
public class SimpleConfigurationBean {

   @Value("#{instanceData.tag1}")
   private String value1;

   @Value("#{instanceData.tag2}")
   private String value2;

   @Value("#{instanceData.tag3}")
   private String value3;

   @Value("#{instanceData.tag4}")
   private String value4;
}
```

5.1.7. Configuring custom EC2 client

In some circumstances it is necessary to have a custom EC2 client to retrieve the instance information. The `context-instance-data` element supports a custom EC2 client with the `amazon-ec2` attribute. The next example shows the use of a custom EC2 client that might have a special configuration in place.

```
<beans ...>

  <aws-context:context-credentials>....</aws-context:context-credentials>
  <aws-context:context-region ... />
  <aws-context:context-instance-data  amazon-ec2="myCustomClient"/>

  <bean id="myCustomClient" class="com.amazonaws.services.ec2.AmazonEC2Client">
    ...
  </bean>
</beans>
```

5.1.8. Injecting the default EC2 client

If there are user tags configured for the instance data (see above) Spring Cloud AWS configures an EC2 client with the specified region and security credentials. Application developers can inject the EC2 client directly into their code using the `@Autowired` annotation.

```
public class ApplicationService {

    private final AmazonEC2 amazonEc2;

    @Autowired
    public ApplicationService(AmazonEC2 amazonEc2) {
        this.amazonEc2 = amazonEc2;
    }
}
```

5.2. Integrating your Spring Cloud application with the AWS Parameter Store

Spring Cloud provides support for centralized configuration, which can be read and made available as a regular Spring `PropertySource` when the application is started. The Parameter Store Configuration allows you to use this mechanism with the AWS Parameter Store.

Simply add a dependency on the `spring-cloud-starter-aws-parameter-store-config` starter module to activate the support. The support is similar to the support provided for the Spring Cloud Config Server or Consul's key-value store: configuration parameters can be defined to be shared across all services or for a specific service and can be profile-specific. Encrypted values will be decrypted when retrieved.

All configuration parameters are retrieved from a common path prefix, which defaults to `/config`. From there shared parameters are retrieved from a path that defaults to `application` and service-specific parameters use a path that defaults to the configured `spring.application.name`. You can use both dots and forward slashes to specify the names of configuration keys. Names of activated profiles will be appended to the path using a separator that defaults to an underscore.

That means that for a service called `my-service` the module by default would find and use these parameters:

parameter key	Spring property	description
`/config/application/cloud.aws.stack.name`	`cloud.aws.stack.name`	Shared by all services that have the Configuration support enabled. Can be overridden with a service- or profile-specific property.
`/config/application_production/cloud.aws.stack.name`	`cloud.aws.stack.name`	Shared by all services that have the Configuration support enabled and have a `production` Spring profile activated. Can be overridden with a service-specific property.

parameter key	Spring property	description
`/config/my-service/cloud/aws/stack/auto`	`cloud.aws.stack.auto`	Specific to the `my-service` service. Note that slashes in the key path are replaced with dots.
`/config/my-service_production/cloud/aws/stack/auto`	`cloud.aws.stack.auto`	Specific to the `my-service` service when a `production` Spring profile is activated.

Note that this module does not support full configuration files to be used as parameter values like e.g. Spring Cloud Consul does: AWS parameter values are limited to 4096 characters, so we support individual Spring properties to be configured only.

You can configure the following settings in a Spring Cloud `bootstrap.properties` or `bootstrap.yml` file (note that relaxed property binding is applied, so you don't have to use this exact syntax):

property	default	explanation
`aws.paramstore.prefix`	`/config`	Prefix indicating first level for every property loaded from the Parameter Store. Value must start with a forward slash followed by one or more valid path segments or be empty.
`aws.paramstore.defaultContext`	`application`	Name of the context that defines properties shared across all services
`aws.paramstore.profileSeparator`	`_`	String that separates an appended profile from the context name. Can only contain dots, dashes, forward slashes, backward slashes and underscores next to alphanumeric characters.
`aws.paramstore.failFast`	`true`	Indicates if an error while retrieving the parameters should fail starting the application.
`aws.paramstore.name`	the configured value for `spring.application.name`	Name to use when constructing the path for the properties to look up for this specific service.
`aws.paramstore.enabled`	`true`	Can be used to disable the Parameter Store Configuration support even though the auto-configuration is on the classpath.

In order to find out which properties are retrieved from AWS Parameter Store on application startup, turn on DEBUG logging on org.springframework.cloud.aws.paramstore.AwsParamStorePropertySource class.

```
logging.level.org.springframework.cloud.aws.paramstore.AwsParamStorePro
pertySource=debug
```

5.3. Integrating your Spring Cloud application with the AWS Secrets Manager

Spring Cloud provides support for centralized configuration, which can be read and made available as a regular Spring PropertySource when the application is started. The Secrets Manager Configuration allows you to use this mechanism with the AWS Secrets Manager.

Simply add a dependency on the spring-cloud-starter-aws-secrets-manager-config starter module to activate the support. The support is similar to the support provided for the Spring Cloud Config Server or Consul's key-value store: configuration parameters can be defined to be shared across all services or for a specific service and can be profile-specific.

All configuration parameters are retrieved from a common path prefix, which defaults to /secret. From there shared parameters are retrieved from a path that defaults to application and service-specific parameters use a path that defaults to the configured spring.application.name. You can use both dots and forward slashes to specify the names of configuration keys. Names of activated profiles will be appended to the path using a separator that defaults to an underscore.

That means that for a service called my-service the module by default would find and use these parameters:

parameter key	description
/secret/application	Shared by all services that have the Configuration support enabled. Can be overridden with a service- or profile-specific property.
/secret/application_production	Shared by all services that have the Configuration support enabled and have a production Spring profile activated. Can be overridden with a service-specific property.
/secret/my-service	Specific to the my-service service..
/secret/my-service_production	Specific to the my-service service when a production Spring profile is activated.

You can configure the following settings in a Spring Cloud bootstrap.properties or bootstrap.yml file (note that relaxed property binding is applied, so you don't have to use this exact syntax):

property	default	explanation
`aws.secretsmanager.prefix`	`/secret`	Prefix indicating first level for every property loaded from the Secrets Manager. Value must start with a forward slash followed by one or more valid path segments or be empty.
`aws.secretsmanager.defaultContext`	`application`	Name of the context that defines properties shared across all services
`aws.secretsmanager.profileSeparator`	`_`	String that separates an appended profile from the context name. Can only contain dots, dashes, forward slashes, backward slashes and underscores next to alphanumeric characters.
`aws.secretsmanager.failFast`	`true`	Indicates if an error while retrieving the secrets should fail starting the application.
`aws.secretsmanager.name`	the configured value for `spring.application.name`	Name to use when constructing the path for the properties to look up for this specific service.
`aws.secretsmanager.enabled`	`true`	Can be used to disable the Secrets Manager Configuration support even though the auto-configuration is on the classpath.

Chapter 6. Managing cloud environments

Managing environments manually with the management console does not scale and can become error-prone with the increasing complexity of the infrastructure. Amazon Web services offers a CloudFormation service that allows to define stack configuration templates and bootstrap the whole infrastructure with the services. In order to allow multiple stacks in parallel, each resource in the stack receives a unique physical name that contains some arbitrary generated name. In order to interact with the stack resources in a unified way Spring Cloud AWS allows developers to work with logical names instead of the random physical ones.

The next graphics shows a typical stack configuration.

[CloudFormation overview] | *cloudformation-overview.png*

The **Template File** describes all stack resources with their *logical name*. The **CloudFormation** service parses the stack template file and creates all resources with their *physical name*. The application can use all the stack configured resources with the *logical name* defined in the template. Spring Cloud AWS resolves all *logical names* into the respective *physical name* for the application developer.

6.1. Automatic CloudFormation configuration

If the application runs inside a stack (because the underlying EC2 instance has been bootstrapped within the stack), then Spring Cloud AWS will automatically detect the stack and resolve all resources from the stack. Application developers can use all the logical names from the stack template to interact with the services. In the example below, the database resource is configured using a CloudFormation template, defining a logical name for the database instance.

```
"applicationDatabase": {
  "Type": "AWS::RDS::DBInstance",
  "Properties": {
    "AllocatedStorage": "5",
    "DBInstanceClass": "db.t1.micro",
    "DBName": "test"
    ...
  ]
 }
}
```

The datasource is then created and will receive a physical name (e.g. ir142c39k6o5irj) as the database service name. Application developers can still use the logical name (in this case `applicationDatabase`) to interact with the database. The example below shows the stack configuration which is defined by the element `aws-context:stack-configuration` and resolves automatically the particular stack. The `data-source` element uses the logical name for the `db-instance-identifier` attribute to work with the database.

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aws-context="http://www.springframework.org/schema/cloud/aws/context"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/context
       http://www.springframework.org/schema/cloud/aws/context/spring-cloud-aws-
context.xsd">

  <aws-context:context-credentials>
    ...
  </aws-context:context-credentials>

  <aws-context:context-region .. />

  <aws-context:stack-configuration/>

  <jdbc:data-source db-instance-identifier="applicationDatabase" ... />
</beans>
```

Further detailed information on the Amazon RDS configuration and setup can be found in the respective chapter in this documentation.

6.2. Manual CloudFormation configuration

If the application is not running inside a stack configured EC2 instance, then the stack configuration must be configured manually. The configuration consists of an additional element attribute `stack-name` that will be used to resolve all the respective stack configuration information at runtime.

```
<beans ....>
    ...
  <aws-context:stack-configuration stack-name="myStackName" />
    ...
</beans>
```

6.3. CloudFormation configuration with Java config classes

Spring Cloud AWS also supports the configuration of the CloudFormation support within Java classes avoiding the use of XML inside the application configuration. Spring Cloud AWS provides the annotation `og.springframework.cloud.aws.context.config.annotation.EnableStackConfiguration` that allows the automatic and manual stack configuration. The next example shows a configuration class that configures the CloudFormation support with an explicit stack name (here `manualStackName`).

```
@Configuration
@EnableStackConfiguration(stackName = "manualStackName")
class ApplicationConfiguration {
}
```

Do not define the `stackName` attribute if an automatic stack name should be enabled.

6.4. CloudFormation configuration in Spring Boot

Spring Cloud AWS also supports the configuration of the CloudFormation support within the Spring Boot configuration. The manual and automatic stack configuration can be defined with properties that are described in the table below.

property	example	description
cloud.aws.stack.name	myStackName	The name of the manually configured stack name that will be used to retrieve the resources.
cloud.aws.stack.auto	true	Enables the automatic stack name detection for the application.

6.5. Manual name resolution

Spring Cloud AWS uses the CloudFormation stack to resolve all resources internally using the logical names. In some circumstances it might be needed to resolve the physical name inside the application code. Spring Cloud AWS provides a pre-configured service to resolve the physical stack name based on the logical name. The sample shows a manual stack resource resolution.

```
@Service
public class ApplicationService {

 private final ResourceIdResolver resourceIdResolver;

 @Autowired
 public ApplicationService(ResourceIdResolver resourceIdResolver) {
    this.resourceIdResolver = resourceIdResolver;
 }

 public void handleApplicationLogic() {
    String physicalBucketName =
        this.resourceIdResolver.resolveToPhysicalResourceId("someLogicalName");
 }
}
```

6.6. Stack Tags

Like for the Amazon EC2 instances, CloudFormation also provides stack specific tags that can be used to configure stack specific configuration information and receive them inside the application. This can for example be a stage specific configuration property (like DEV, INT, PRD).

```
<beans ....>
    ...
    <aws-context:stack-configuration user-tags-map="stackTags"/>
    ...
</beans>
```

The application can then access the stack tags with an expression like `#{stackTags.key1}`.

6.7. Using custom CloudFormation client

Like for the EC2 configuration setup, the `aws-context:stack-configuration` element supports a custom CloudFormation client with a special setup. The client itself can be configured using the `amazon-cloud-formation` attribute as shown in the example:

```
<beans>
    <aws-context:stack-configuration amazon-cloud-formation=""/>

    <bean class="com.amazonaws.services.cloudformation.AmazonCloudFormationClient">
    </bean>
</beans>
```

Chapter 7. Messaging

Spring Cloud AWS provides Amazon SQS and Amazon SNS integration that simplifies the publication and consumption of messages over SQS or SNS. While SQS fully relies on the messaging API introduced with Spring 4.0, SNS only partially implements it as the receiving part must be handled differently for push notifications.

7.1. Configuring messaging

Before using and configuring the messaging support, the application has to include the respective module dependency into the Maven configuration. Spring Cloud AWS Messaging support comes as a separate module to allow the modularized use of the modules.

7.1.1. Maven dependency configuration

The Spring Cloud AWS messaging module comes as a standalone module and can be imported with the following dependency declaration:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-aws-messaging</artifactId>
    <version>{spring-cloud-version}</version>
</dependency>
```

7.2. SQS support

Amazon SQS is a hosted messaging service on the Amazon Web Service platform that provides point-to-point communication with queues. Compared to JMS or other message services Amazon SQS has several features and limitations that should be taken into consideration.

- Amazon SQS allows only `String` payloads, so any `Object` must be transformed into a String representation. Spring Cloud AWS has dedicated support to transfer Java objects with Amazon SQS messages by converting them to JSON.
- Amazon SQS has no transaction support, so messages might therefore be retrieved twice. Application have to be written in an idempotent way so that they can receive a message twice.
- Amazon SQS has a maximum message size of 256kb per message, so bigger messages will fail to be sent.

7.2.1. Sending a message

The `QueueMessagingTemplate` contains many convenience methods to send a message. There are send methods that specify the destination using a `QueueMessageChannel` object and those that specify the destination using a string which is going to be resolved against the SQS API. The send method that takes no destination argument uses the default destination.

```
import com.amazonaws.services.sqs.AmazonSQSAsync;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.aws.messaging.core.QueueMessagingTemplate;
import org.springframework.messaging.support.MessageBuilder;

public class SqsQueueSender {

    private final QueueMessagingTemplate queueMessagingTemplate;

    @Autowired
    public SqsQueueSender(AmazonSQSAsync amazonSQSAsync) {
        this.queueMessagingTemplate = new QueueMessagingTemplate(amazonSQSAsync);
    }

    public void send(String message) {
        this.queueMessagingTemplate.send("physicalQueueName",
MessageBuilder.withPayload(message).build());
    }
}
```

This example uses the `MessageBuilder` class to create a message with a string payload. The `QueueMessagingTemplate` is constructed by passing a reference to the `AmazonSQSAsync` client. The destination in the send method is a string value that must match the queue name defined on AWS. This value will be resolved at runtime by the Amazon SQS client. Optionally a `ResourceIdResolver` implementation can be passed to the `QueueMessagingTemplate` constructor to resolve resources by logical name when running inside a CloudFormation stack (see Managing cloud environments for more information about resource name resolution).

With the messaging namespace a `QueueMessagingTemplate` can be defined in an XML configuration file.

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aws-context="http://www.springframework.org/schema/cloud/aws/context"
    xmlns:aws-messaging="http://www.springframework.org/schema/cloud/aws/messaging"
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cloud/aws/context
        http://www.springframework.org/schema/cloud/aws/context/spring-cloud-aws-
context.xsd
        http://www.springframework.org/schema/cloud/aws/messaging
        http://www.springframework.org/schema/cloud/aws/messaging/spring-cloud-aws-
messaging">

    <aws-context:context-credentials>
        <aws-context:instance-profile-credentials />
    </aws-context:context-credentials>

    <aws-messaging:queue-messaging-template id="queueMessagingTemplate" />

</beans>
```

In this example the messaging namespace handler constructs a new `QueueMessagingTemplate`. The `AmazonSQSAsync` client is automatically created and passed to the template's constructor based on the provided credentials. If the application runs inside a configured CloudFormation stack a `ResourceIdResolver` is passed to the constructor (see Managing cloud environments for more information about resource name resolution).

Using message converters

In order to facilitate the sending of domain model objects, the `QueueMessagingTemplate` has various send methods that take a Java object as an argument for a message's data content. The overloaded methods `convertAndSend()` and `receiveAndConvert()` in `QueueMessagingTemplate` delegate the conversion process to an instance of the `MessageConverter` interface. This interface defines a simple contract to convert between Java objects and SQS messages. The default implementation `SimpleMessageConverter` simply unwraps the message payload as long as it matches the target type. By using the converter, you and your application code can focus on the business object that is being sent or received via SQS and not be concerned with the details of how it is represented as an SQS message.

As SQS is only able to send `String` payloads the default converter `SimpleMessageConverter` should only be used to send `String` payloads. For more complex objects a custom converter should be used like the one created by the messaging namespace handler.

It is recommended to use the XML messaging namespace to create `QueueMessagingTemplate` as it will set a more sophisticated `MessageConverter` that converts objects into JSON when Jackson is on the classpath.

```
<aws-messaging:queue-messaging-template id="queueMessagingTemplate" />
```

```
this.queueMessagingTemplate.convertAndSend("queueName", new Person("John, "Doe"));
```

In this example a `QueueMessagingTemplate` is created using the messaging namespace. The `convertAndSend` method converts the payload `Person` using the configured `MessageConverter` and sends the message.

7.2.2. Receiving a message

There are two ways for receiving SQS messages, either use the `receive` methods of the `QueueMessagingTemplate` or with annotation-driven listener endpoints. The latter is by far the more convenient way to receive messages.

```
Person person = this.queueMessagingTemplate.receiveAndConvert("queueName",
Person.class);
```

In this example the `QueueMessagingTemplate` will get one message from the SQS queue and convert it to the target class passed as argument.

7.2.3. Annotation-driven listener endpoints

Annotation-driven listener endpoints are the easiest way for listening on SQS messages. Simply annotate methods with `MessageMapping` and the `QueueMessageHandler` will route the messages to the annotated methods.

```
<aws-messaging:annotation-driven-queue-listener />
```

```
@SqsListener("queueName")
public void queueListener(Person person) {
    // ...
}
```

In this example a queue listener container is started that polls the SQS `queueName` passed to the `MessageMapping` annotation. The incoming messages are converted to the target type and then the annotated method `queueListener` is invoked.

In addition to the payload, headers can be injected in the listener methods with the `@Header` or `@Headers` annotations. `@Header` is used to inject a specific header value while `@Headers` injects a `Map<String, String>` containing all headers.

Only the standard message attributes sent with an SQS message are supported. Custom attributes are currently not supported.

In addition to the provided argument resolvers, custom ones can be registered on the `aws-messaging:annotation-driven-queue-listener` element using the `aws-messaging:argument-resolvers` attribute (see example below).

```
<aws-messaging:annotation-driven-queue-listener>
    <aws-messaging:argument-resolvers>
        <bean class="org.custom.CustomArgumentResolver" />
    </aws-messaging:argument-resolvers>
</aws-messaging:annotation-driven-queue-listener>
```

By default the `SimpleMessageListenerContainer` creates a `ThreadPoolTaskExecutor` with computed values for the core and max pool sizes. The core pool size is set to twice the number of queues and the max pool size is obtained by multiplying the number of queues by the value of the `maxNumberOfMessages` field. If these default values do not meet the need of the application, a custom task executor can be set with the `task-executor` attribute (see example below).

```
<aws-messaging:annotation-driven-queue-listener task-executor="simpleTaskExecutor" />
```

Message reply

Message listener methods can be annotated with `@SendTo` to send their return value to another channel. The `SendToHandlerMethodReturnValueHandler` uses the defined messaging template set on the `aws-messaging:annotation-driven-queue-listener` element to send the return value. The messaging template must implement the `DestinationResolvingMessageSendingOperations` interface.

```
<aws-messaging:annotation-driven-queue-listener send-to-message-
template="queueMessagingTemplate"/>
```

```
@SqsListener("treeQueue")
@SendTo("leafsQueue")
public List<Leaf> extractLeafs(Tree tree) {
    // ...
}
```

In this example the `extractLeafs` method will receive messages coming from the `treeQueue` and then return a `List` of `Leaf`s which is going to be sent to the `leafsQueue`. Note that on the `aws-messaging:annotation-driven-queue-listener` XML element there is an attribute `send-to-message-template` that specifies `QueueMessagingTemplate` as the messaging template to be used to send the return value of the message listener method.

Handling Exceptions

Exception thrown inside `@SqsListener` annotated methods can be handled by methods annotated with `@MessageExceptionHandler`.

```
import org.springframework.cloud.aws.messaging.listener.annotation.SqsListener;
import org.springframework.messaging.handler.annotation.MessageExceptionHandler;
import org.springframework.stereotype.Component;

@Component
public class MyMessageHandler {

    @SqsListener("queueName")
    void handle(String message) {
        ...
        throw new MyException("something went wrong");
    }

    @MessageExceptionHandler(MyException.class)
    void handleException(MyException e) {
        ...
    }
}
```

7.2.4. The SimpleMessageListenerContainerFactory

The `SimpleMessageListenerContainer` can also be configured with Java by creating a bean of type `SimpleMessageListenerContainerFactory`.

```
@Bean
public SimpleMessageListenerContainerFactory
simpleMessageListenerContainerFactory(AmazonSQSAsync amazonSqs) {
    SimpleMessageListenerContainerFactory factory = new
SimpleMessageListenerContainerFactory();
    factory.setAmazonSqs(amazonSqs);
    factory.setAutoStartup(false);
    factory.setMaxNumberOfMessages(5);
    // ...

    return factory;
}
```

7.2.5. Consuming AWS Event messages with Amazon SQS

It is also possible to receive AWS generated event messages with the SQS message listeners. Because AWS messages does not contain the mime-type header, the Jackson message converter has to be configured with the `strictContentTypeMatch` property false to also parse message without the proper mime type.

The next code shows the configuration of the message converter using the `QueueMessageHandlerFactory` and re-configuring the `MappingJackson2MessageConverter`

```
@Bean
public QueueMessageHandlerFactory queueMessageHandlerFactory() {
    QueueMessageHandlerFactory factory = new QueueMessageHandlerFactory();
    MappingJackson2MessageConverter messageConverter = new
MappingJackson2MessageConverter();

    //set strict content type match to false
    messageConverter.setStrictContentTypeMatch(false);

factory.setArgumentResolvers(Collections.<HandlerMethodArgumentResolver>singletonList(
new PayloadArgumentResolver(messageConverter)));
    return factory;
}
```

With the configuration above, it is possible to receive event notification for S3 buckets (and also other event notifications like elastic transcoder messages) inside `@SqsListener` annotated methods s shown below.

```
@SqsListener("testQueue")
public void receive(S3EventNotification s3EventNotificationRecord) {
    S3EventNotification.S3Entity s3Entity =
s3EventNotificationRecord.getRecords().get(0).getS3();
}
```

7.3. SNS support

Amazon SNS is a publish-subscribe messaging system that allows clients to publish notification to a particular topic. Other interested clients may subscribe using different protocols like HTTP/HTTPS, e-mail or an Amazon SQS queue to receive the messages.

The next graphic shows a typical example of an Amazon SNS architecture.

[SNS Overview] | *sns-overview.png*

Spring Cloud AWS supports Amazon SNS by providing support to send notifications with a `NotificationMessagingTemplate` and to receive notifications with the HTTP/HTTPS endpoint using the Spring Web MVC `@Controller` based programming model. Amazon SQS based subscriptions can be used with the annotation-driven message support that is provided by the Spring Cloud AWS messaging module.

7.3.1. Sending a message

The `NotificationMessagingTemplate` contains two convenience methods to send a notification. The first one specifies the destination using a `String` which is going to be resolved against the SNS API. The second one takes no destination argument and uses the default destination. All the usual send methods that are available on the `MessageSendingOperations` are implemented but are less convenient to send notifications because the subject must be passed as header.

Currently only `String` payloads can be sent using the `NotificationMessagingTemplate` as this is the expected type by the SNS API.

```
import com.amazonaws.services.sns.AmazonSNS;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.aws.messaging.core.NotificationMessagingTemplate;

public class SnsNotificationSender {

    private final NotificationMessagingTemplate notificationMessagingTemplate;

    @Autowired
    public SnsNotificationSender(AmazonSNS amazonSns) {
        this.notificationMessagingTemplate = new
NotificationMessagingTemplate(amazonSns);
    }

    public void send(String subject, String message) {
        this.notificationMessagingTemplate.sendNotification("physicalTopicName",
message, subject);
    }
}
```

This example constructs a new `NotificationMessagingTemplate` by passing an `AmazonSNS` client as argument. In the `send` method the convenience `sendNotification` method is used to send a `message` with `subject` to an SNS topic. The destination in the `sendNotification` method is a string value that must match the topic name defined on AWS. This value is resolved at runtime by the Amazon SNS client. Optionally a `ResourceIdResolver` implementation can be passed to the `NotificationMessagingTemplate` constructor to resolve resources by logical name when running inside a CloudFormation stack. (See Managing cloud environments for more information about resource name resolution.)

It is recommended to use the XML messaging namespace to create `NotificationMessagingTemplate` as it will automatically configure the SNS client to setup the default converter.

```
<aws-messaging:notification-messaging-template id="notificationMessagingTemplate" />
```

7.3.2. Annotation-driven HTTP notification endpoint

SNS supports multiple endpoint types (SQS, Email, HTTP, HTTPS), Spring Cloud AWS provides support for HTTP(S) endpoints. SNS sends three type of requests to an HTTP topic listener endpoint, for each of them annotations are provided:

- Subscription request → `@NotificationSubscriptionMapping`
- Notification request → `@NotificationMessageMapping`
- Unsubscription request → `@NotificationUnsubscribeMapping`

HTTP endpoints are based on Spring MVC controllers. Spring Cloud AWS added some custom argument resolvers to extract the message and subject out of the notification requests.

```
@Controller
@RequestMapping("/topicName")
public class NotificationTestController {

    @NotificationSubscriptionMapping
    public void handleSubscriptionMessage(NotificationStatus status) throws
IOException {
        //We subscribe to start receive the message
        status.confirmSubscription();
    }

    @NotificationMessageMapping
    public void handleNotificationMessage(@NotificationSubject String subject,
@NotificationMessage String message) {
        // ...
    }

    @NotificationUnsubscribeConfirmationMapping
    public void handleUnsubscribeMessage(NotificationStatus status) {
        //e.g. the client has been unsubscribed and we want to "re-subscribe"
        status.confirmSubscription();
    }
}
```

Currently it is not possible to define the mapping URL on the method level therefore the `RequestMapping` must be done at type level and must contain the full path of the endpoint.

This example creates a new Spring MVC controller with three methods to handle the three requests listed above. In order to resolve the arguments of the `handleNotificationMessage` methods a custom argument resolver must be registered. The XML configuration is listed below.

```
<mvc:annotation-driven>
    <mvc:argument-resolvers>
        <ref bean="notificationResolver" />
    </mvc:argument-resolvers>
</mvc:annotation-driven>

<aws-messaging:notification-argument-resolver id="notificationResolver" />
```

The `aws-messaging:notification-argument-resolver` element registers three argument resolvers: `NotificationStatusHandlerMethodArgumentResolver`, `NotificationMessageHandlerMethodArgumentResolver`, and `NotificationSubjectHandlerMethodArgumentResolver`.

7.4. Using CloudFormation

Amazon SQS queues and SNS topics can be configured within a stack and then be used by applications. Spring Cloud AWS also supports the lookup of stack-configured queues and topics by their logical name with the resolution to the physical name. The example below shows an SNS topic and SQS queue configuration inside a CloudFormation template.

```
"LogicalQueueName": {
    "Type": "AWS::SQS::Queue",
    "Properties": {
    }
},
"LogicalTopicName": {
    "Type": "AWS::SNS::Topic",
    "Properties": {
    }
}
```

The logical names `LogicalQueueName` and `LogicalTopicName` can then be used in the configuration and in the application as shown below:

```
<aws-messaging:queue-messaging-template default-destination="LogicalQueueName" />

<aws-messaging:notification-messaging-template default-destination="LogicalTopicName"
/>
```

```
@SqsListener("LogicalQueueName")
public void receiveQueueMessages(Person person) {
    // Logical names can also be used with messaging templates
    this.notificationMessagingTemplate.sendNotification("anotherLogicalTopicName",
"Message", "Subject");
}
```

When using the logical names like in the example above, the stack can be created on different environments without any configuration or code changes inside the application.

Chapter 8. Caching

Caching in a cloud environment is useful for applications to reduce the latency and to save database round trips. Reducing database round trips can significantly reduce the requirements for the database instance. The Spring Framework provides, since version 3.1, a unified Cache abstraction to allow declarative caching in applications analogous to the declarative transactions.

Spring Cloud AWS integrates the Amazon ElastiCache service into the Spring unified caching abstraction providing a cache manager based on the memcached and Redis protocols. The caching support for Spring Cloud AWS provides its own memcached implementation for ElastiCache and uses Spring Data Redis for Redis caches.

8.1. Configuring dependencies for Redis caches

Spring Cloud AWS delivers its own implementation of a memcached cache, therefore no other dependencies are needed. For Redis Spring Cloud AWS relies on Spring Data Redis to support caching and also to allow multiple Redis drivers to be used. Spring Cloud AWS supports all Redis drivers that Spring Data Redis supports (currently Jedis, JRedis, SRP and Lettuce) with Jedis being used internally for testing against ElastiCache. A dependency definition for Redis with Jedis is shown in the example

```
<dependencies>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-redis</artifactId>
        <version>${spring-data-redis.version}</version>
    </dependency>
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>2.6.1</version>
    </dependency>
</dependencies>
```

Spring Cloud AWS will automatically detect the Redis driver and will use one of them automatically.

8.2. Configuring caching with XML

The cache support for Spring Cloud AWS resides in the context module and can therefore be used if the context module is already imported in the project. The cache integration provides its own namespace to configure cache clusters that are hosted in the Amazon ElastiCache service. The next example contains a configuration for the cache cluster and the Spring configuration to enable declarative, annotation-based caching.

```
<beans xmlns:aws-cache="http://www.springframework.org/schema/cloud/aws/cache"
       xmlns:cache="http://www.springframework.org/schema/cache"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/cache
        http://www.springframework.org/schema/cloud/aws/cache/spring-cloud-aws-
cache.xsd
        http://www.springframework.org/schema/cache
        https://www.springframework.org/schema/cache/spring-cache.xsd">

    <aws-context:context-credentials>
        ...
        </aws-context:context-credentials>

    <aws-cache:cache-manager>
        <aws-cache:cache-cluster name="CacheCluster" />
    </aws-cache:cache-manager>

    <cache:annotation-driven />
</beans>
```

The configuration above configures a `cache-manager` with one cache with the name `CacheCluster` that represents an ElasticCache cluster.

8.2.1. Mixing caches

Applications may have the need for multiple caches that are maintained by one central cache cluster. The Spring Cloud AWS caching support allows to define multiple caches inside one cache manager and also to use externally defined caches inside the cache manager.

The example below demonstrates a configuration example that contains a pre-configured cache with a `cache-ref` element (which might be a local cache) and a `cache-cluster` configuration for ElastiCache cache clusters.

```
<beans ...>
    <aws-cache:cache-manager id="cacheManager">
        <aws-cache:cache-ref ref="memcached" />
        <aws-cache:cache-cluster name="SimpleCache"/>
    </aws-cache:cache-manager>
</beans>
```

8.2.2. Defining expiration

The Spring cache demarcation does not support expiry time configuration and leaves it up to the cache implementation to support an expiry time. The Spring Cloud AWS cache configuration supports the expiry time setting per cache. The expiry time will be passed to the memcached service.

The `cache-cluster` element accepts an expiration attribute that defines the expiration time in

seconds. No configured values implies that there is an infinite expiration time.

```
<beans>
    <aws-cache:cache-manager>
        <aws-cache:cache-cluster expiration="10000" name="CacheCluster" />
    </aws-cache:cache-manager>
</beans>
```

8.3. Configuring caching using Java configuration

Spring Cloud AWS also support the cache configuration with Java configuration classes. On any `Configuration` class, the caching can be configured using the `org.springframework.cloud.aws.cache.config.annotation.EnableElastiCache` annotation provided by Spring Cloud AWS. The next example shows a configuration of two cache clusters.

```
@EnableElastiCache({@CacheClusterConfig(name = "firstCache"), @CacheClusterConfig(name
= "secondCache")})
public class ApplicationConfiguration {
}
```

If you leave the `value` attribute empty, then all the caches inside your CloudFormation stack (if available) will be configured automatically.

8.3.1. Configuring expiry time for caches

The Java configuration also allows to configure the expiry time for the caches. This can be done for all caches using the `defaultExpiration` attribute as shown in the example below.

```
@EnableElastiCache(defaultExpiration = 23)
public class ApplicationConfiguration {
}
```

The expiration can be defined on a cache level using the `@CacheClusterConfig` annotations expiration attribute as shown below (using seconds as the value).

```
@EnableElastiCache({@CacheClusterConfig(name = "firstCache", expiration = 23),
@CacheClusterConfig(name = "secondCache", expiration = 42)})
public class ApplicationConfiguration {
}
```

8.4. Configuring caching in Spring Boot

The caches will automatically be configured in Spring Boot without any explicit configuration property.

8.5. Using caching

Based on the configuration of the cache, developers can annotate their methods to use the caching for method return values. The next example contains a caching declaration for a service for which the return values should be cached

```
@Service
public class ExpensiveService {

    @Cacheable("CacheCluster")
    public String calculateExpensiveValue(String key) {
        ...
    }
}
```

8.6. Memcached client implementation

There are different memcached client implementations available for Java, the most prominent ones are Spymemcached and XMemcached. Amazon AWS supports a dynamic configuration and delivers an enhanced memcached client based on Spymemcached to support the auto-discovery of new nodes based on a central configuration endpoint.

Spring Cloud AWS relies on the Amazon ElastiCache Client implementation and therefore has a dependency on that.

8.7. Using CloudFormation

Amazon ElastiCache clusters can also be configured within a stack and then be used by applications. Spring Cloud AWS also supports the lookup of stack-configured cache clusters by their logical name with the resolution to the physical name. The example below shows a cache cluster configuration inside a CloudFormation template.

```
"CacheCluster": {
    "Type": "AWS::ElastiCache::CacheCluster",
    "Properties": {
        "AutoMinorVersionUpgrade": "true",
        "Engine": "memcached",
        "CacheNodeType": "cache.t2.micro",
        "CacheSubnetGroupName" : "sample",
        "NumCacheNodes": "1",
        "VpcSecurityGroupIds": ["sample1"]
    }
}
```

The cache cluster can then be used with the name `CacheCluster` inside the application configuration as shown below:

```
<beans...>
    <aws-cache:cache-manager>
        <aws-cache:cache-cluster name="CacheCluster" expiration="15"/>
    </aws-cache:cache-manager>
<beans>
```

With the configuration above the application can be deployed with multiple stacks on different environments without any configuration change inside the application.

Chapter 9. Data Access with JDBC

Spring has a broad support of data access technologies built on top of JDBC like `JdbcTemplate` and dedicated ORM (JPA, Hibernate support). Spring Cloud AWS enables application developers to re-use their JDBC technology of choice and access the Amazon Relational Database Service with a declarative configuration. The main support provided by Spring Cloud AWS for JDBC data access are:

- Automatic data source configuration and setup based on the Amazon RDS database instance.
- Automatic read-replica detection and configuration for Amazon RDS database instances.
- Retry-support to handle exception during Multi-AZ failover inside the data center.

9.1. Configuring data source

Before using and configuring the database support, the application has to include the respective module dependency into its Maven configuration. Spring Cloud AWS JDBC support comes as a separate module to allow the modularized use of the modules.

9.1.1. Maven dependency configuration

The Spring Cloud AWS JDBC module comes as a standalone module and can be imported with the following dependency declaration.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-aws-jdbc</artifactId>
    <version>{spring-cloud-version}</version>
</dependency>
```

9.1.2. Basic data source configuration

The data source configuration requires the security and region configuration as a minimum allowing Spring Cloud AWS to retrieve the database metadata information with the Amazon RDS service. Spring Cloud AWS provides an additional `jdbc` specific namespace to configure the data source with the minimum attributes as shown in the example:

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jdbc="http://www.springframework.org/schema/cloud/aws/jdbc"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/jdbc
       http://www.springframework.org/schema/cloud/aws/jdbc/spring-cloud-aws-
jdbc.xsd">

 <aws-context:context-credentials>
  ...
 </aws-context:context-credentials>

 <aws-context:context-region region="..."/>

<jdbc:data-source
         db-instance-identifier="myRdsDatabase"
         password="${rdsPassword}">
</jdbc:data-source>
</beans>
```

The minimum configuration parameters are a unique `id` for the data source, a valid `db-instance-identifier` attribute that points to a valid Amazon RDS database instance. The master user password for the master user. If there is another user to be used (which is recommended) then the `username` attribute can be set.

With this configuration Spring Cloud AWS fetches all the necessary metadata and creates a Tomcat JDBC pool with the default properties. The data source can be later injected into any Spring Bean as shown below:

```
@Service
public class SimpleDatabaseService implements DatabaseService {

 private final JdbcTemplate jdbcTemplate;

 @Autowired
 public SimpleDatabaseService(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
 }
}
```

It is possible to qualify the data source injection point with an `@Qualifier` annotation to allow multiple data source configurations inside one application context and still use auto-wiring.

9.1.3. Data source pool configuration

Spring Cloud AWS creates a new Tomcat JDBC pool with the default properties. Often these default properties do not meet the requirements of the application with regards to pool size and other settings. The data source configuration supports the configuration of all valid pool properties with a nested XML element. The following example demonstrates the re-configuration of the data source

with custom pool properties.

```
<beans ..>

 <aws-context:context-credentials>
  ...
 </aws-context:context-credentials>

 <aws-context:context-region region="..."/>

 <jdbc:data-source
         db-instance-identifier="myRdsDatabase"
         password="${rdsPassword}">
         <jdbc:pool-attributes initialSize="1" " maxActive="200" minIdle="10"
                testOnBorrow="true" validationQuery="SELECT 1" />
 </jdbc:data-source>

</beans>
```

A full list of all configuration attributes with their value is available here.

9.2. Configuring data source with Java config

Spring Cloud AWS also supports the configuration of the data source within an `@Configuration` class. The `org.springframework.cloud.aws.jdbc.config.annotation.EnableRdsInstance` annotation can be used to configure one data source. Multiple ones can be used to configure more then one data source. Each annotation will generate exactly one data source bean.

The class below shows a data source configuration inside a configuration class

```
@Configuration
@EnableRdsInstance(dbInstanceIdentifier = "test",password = "secret",
readReplicaSupport = true)
public class ApplicationConfiguration {
}
```

The configuration attributes are the same in the XML element. The required attributes are also the same for the XML configuration (the `dbInstanceIdentifier` and `password` attribute)

9.2.1. Java based data source pool configuration

It is also possible to override the pool configuration with custom values. Spring Cloud AWS provides a `org.springframework.cloud.aws.jdbc.config.annotation.RdsInstanceConfigurer` that creates a `org.springframework.cloud.aws.jdbc.datasource.DataSourceFactory` which might contain custom pool attributes. The next examples shows the implementation of one configurer that overrides the validation query and the initial size.

```
@Configuration
@EnableRdsInstance(dbInstanceIdentifier = "test",password = "secret")
public class ApplicationConfiguration {

    @Bean
    public RdsInstanceConfigurer instanceConfigurer() {
        return new RdsInstanceConfigurer() {
            @Override
            public DataSourceFactory getDataSourceFactory() {
                TomcatJdbcDataSourceFactory dataSourceFactory = new
TomcatJdbcDataSourceFactory();
                dataSourceFactory.setInitialSize(10);
                dataSourceFactory.setValidationQuery("SELECT 1 FROM DUAL");
                return dataSourceFactory;
            }
        };
    }
}
```

This class returns an anonymous class of type `org.springframework.cloud.aws.jdbc.config.annotation.RdsInstanceConfigurer`, which might also of course be a standalone class.

9.3. Configuring data source in Spring Boot

The data sources can also be configured using the Spring Boot configuration files. Because of the dynamic number of data sources inside one application, the Spring Boot properties must be configured for each data source.

A data source configuration consists of the general property name `cloud.aws.rds.<instanceName>` for the data source identifier following the sub properties for the particular data source where `instanceName` is the name of the concrete instance. The table below outlines all properties for a data source using `test` as the instance identifier.

property	example	description
cloud.aws.rds.test.password	verySecret	The password for the db instance test
cloud.aws.rds.test.username	admin	The username for the db instance test (optional)
cloud.aws.rds.test.readReplicaSupport	true	If read-replicas should be used for the data source (see below)
cloud.aws.rds.test.databaseName	fooDb	Custom database name if the default one from rds should not be used

9.4. Read-replica configuration

Amazon RDS allows to use MySQL, MariaDB, Oracle, PostgreSQL and Microsoft SQL Server read-replica instances to increase the overall throughput of the database by offloading read data access to one or more read-replica slaves while maintaining the data in one master database.

Spring Cloud AWS supports the use of read-replicas in combination with Spring read-only transactions. If the read-replica support is enabled, any read-only transaction will be routed to a read-replica instance while using the master database for write operations.

Using read-replica instances does not guarantee strict ACID semantics for the database access and should be used with care. This is due to the fact that the read-replica might be behind and a write might not be immediately visible to the read transaction. Therefore it is recommended to use read-replica instances only for transactions that read data which is not changed very often and where outdated data can be handled by the application.

The read-replica support can be enabled with the `read-replica` attribute in the datasource configuration.

```
<beans ..>
 <jdbc:data-source db-instance-identifier="RdsSingleMicroInstance"
    password="${rdsPassword}" read-replica-support="true">

 </jdbc:data-source>
</beans>
```

Spring Cloud AWS will search for any read-replica that is created for the master database and route the read-only transactions to one of the read-replicas that are available. A business service that uses read-replicas can be implemented like shown in the example.

```java
@Service
public class SimpleDatabaseService {

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public SimpleDatabaseService(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Transactional(readOnly = true)
    public Person loadAll() {
        // read data on the read replica
    }

    @Transactional
    public void updatePerson(Person person) {
        // write data into database
    }
}
```

9.5. Failover support

Amazon RDS supports a Multi-AZ fail-over if one availability zone is not available due to an outage or failure of the primary instance. The replication is synchronous (compared to the read-replicas) and provides continuous service. Spring Cloud AWS supports a Multi-AZ failover with a retry mechanism to recover transactions that fail during a Multi-AZ failover.

In most cases it is better to provide direct feedback to a user instead of trying potentially long and frequent retries within a user interaction. Therefore the fail-over support is primarily useful for batch application or applications where the responsiveness of a service call is not critical.

The Spring Cloud AWS JDBC module provides a retry interceptor that can be used to decorate services with an interceptor. The interceptor will retry the database operation again if there is a temporary error due to a Multi-AZ failover. A Multi-AZ failover typically lasts only a couple of seconds, therefore a retry of the business transaction will likely succeed.

The interceptor can be configured as a regular bean and then be used by a pointcut expression to decorate the respective method calls with the interceptor. The interceptor must have a configured database to retrieve the current status (if it is a temporary fail-over or a permanent error) from the Amazon RDS service.

The configuration for the interceptor can be done with a custom element from the Spring Cloud AWS jdbc namespace and will be configured like shown:

```
<beans ..>
    <jdbc:retry-interceptor id="myInterceptor"
        db-instance-identifier="myRdsDatabase"
        max-number-of-retries="10" />
</beans>
```

The interceptor itself can be used with any Spring advice configuration to wrap the respective service. A pointcut for the services shown in the chapter before can be defined as follows:

```
<beans ..>
 <aop:config>
  <aop:advisor advice-ref="myInterceptor" pointcut="bean(simpleDatabaseService)"
order="1" />
 </aop:config>
</beans>
```

It is important that the interceptor is called outside the transaction interceptor to ensure that the whole transaction will be re-executed. Configuring the interceptor inside the transaction interceptor will lead to a permanent error because the broken connection will never be refreshed.

The configuration above in combination with a transaction configuration will produce the following proxy configuration for the service.

[Retry interceptor] | *jdbc-retry-interceptor.png*

9.6. CloudFormation support

Spring Cloud AWS supports database instances that are configured with CloudFormation. Spring Cloud AWS can use the logical name inside the database configuration and lookup the concrete database with the generated physical resource name. A database configuration can be easily configured in CloudFormation with a template definition that might look like the following example.

```
"myRdsDatabase": {
    "Type": "AWS::RDS::DBInstance",
    "Properties": {
     "AllocatedStorage": "5",
     "DBInstanceClass": "db.t1.micro",
     "DBName": "test",
     "Engine": "mysql",
     "MasterUsername": "admin",
     "MasterUserPassword": {"Ref":"RdsPassword"},
     ...
    }
 },
 "readReplicaDatabase": {
    "Type": "AWS::RDS::DBInstance",
    "Properties": {
     "AllocatedStorage" : "5",
     "SourceDBInstanceIdentifier": {
        "Ref": "myRdsDatabase"
     },
     "DBInstanceClass": "db.t1.micro"
    }
 }
}
```

The database can then be configured using the name set in the template. Also, the read-replica can be enabled to use the configured read-replica database in the application. A configuration to use the configured database is outlined below:

```
<beans>
 <aws-context:stack-configuration/>

 <jdbc:data-source db-instance-identifier="myRdsDatabase" password="${rdsPassword}"
read-replica-support="true"/>
</beans>
```

9.7. Database tags

Amazon RDS instances can also be configured using RDS database specific tags, allowing users to configure database specific configuration metadata with the database. Database instance specific tags can be configured using the `user-tags-map` attribute on the `data-source` element. Configure the tags support like in the example below:

```
<jdbc:data-source
    db-instance-identifier="myRdsDatabase"
    password="${rdsPassword}" user-tags-map="dbTags" />
```

That allows the developer to access the properties in the code using expressions like shown in the class below:

```
public class SampleService {

    @Value("#{dbTags['aws:cloudformation:aws:cloudformation:stack-name']}")
    private String stackName;
}
```

The database tag `aws:cloudformation:aws:cloudformation:stack-name` is a default tag that is created if the database is configured using CloudFormation.

Chapter 10. Sending mails

Spring has a built-in support to send e-mails based on the Java Mail API to avoid any static method calls while using the Java Mail API and thus supporting the testability of an application. Spring Cloud AWS supports the Amazon SES as an implementation of the Spring Mail abstraction.

As a result Spring Cloud AWS users can decide to use the Spring Cloud AWS implementation of the Amazon SES service or use the standard Java Mail API based implementation that sends e-mails via SMTP to Amazon SES.

It is preferred to use the Spring Cloud AWS implementation instead of SMTP mainly for performance reasons. Spring Cloud AWS uses one API call to send a mail message, while the SMTP protocol makes multiple requests (EHLO, MAIL FROM, RCPT TO, DATA, QUIT) until it sends an e-mail.

10.1. Configuring the mail sender

Spring Cloud AWS provides an XML element to configure a Spring `org.springframework.mail.MailSender` implementation for the client to be used. The default mail sender works without a Java Mail dependency and is capable of sending messages without attachments as simple mail messages. A configuration with the necessary elements will look like this:

```
<beans xmlns:aws-mail="http://www.springframework.org/schema/cloud/aws/mail"
   xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/mail
      http://www.springframework.org/schema/cloud/aws/mail/spring-cloud-aws-mail.xsd">

    <aws-context:context-credentials>
      ..
    </aws-context:context-credentials>

    <aws-context:context-region region="eu-west-1" />

    <aws-mail:mail-sender id="testSender" />

</beans>
```

10.2. Sending simple mails

Application developers can inject the `MailSender` into their application code and directly send simple text based e-mail messages. The sample below demonstrates the creation of a simple mail message.

```
public class MailSendingService {

    private MailSender mailSender;

    @Autowired
    public MailSendingService(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void sendMailMessage() {
        SimpleMailMessage simpleMailMessage = new SimpleMailMessage();
        simpleMailMessage.setFrom("foo@bar.com");
        simpleMailMessage.setTo("bar@baz.com");
        simpleMailMessage.setSubject("test subject");
        simpleMailMessage.setText("test content");
        this.mailSender.send(simpleMailMessage);
    }
}
```

10.3. Sending attachments

Sending attachments with e-mail requires MIME messages to be created and sent. In order to create MIME messages, the Java Mail dependency is required and has to be included in the classpath. Spring Cloud AWS will detect the dependency and create a `org.springframework.mail.javamail.JavaMailSender` implementation that allows to create and build MIME messages and send them. A dependency configuration for the Java Mail API is the only change in the configuration which is shown below.

```
<dependency>
    <groupId>javax.mail</groupId>
    <artifactId>mailapi</artifactId>
    <version>1.4.1</version>
    <exclusions>
        <!-- exclusion because we are running on Java 1.7 that includes the activation
API by default-->
        <exclusion>
            <artifactId>activation</artifactId>
            <groupId>javax.activation</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

Even though there is a dependency to the Java Mail API there is still the Amazon SES API used underneath to send mail messages. There is no SMTP setup required on the Amazon AWS side.

Sending the mail requires the application developer to use the `JavaMailSender` to send an e-mail as

shown in the example below.

```
public class MailSendingService {

    private JavaMailSender mailSender;

    @Autowired
    public MailSendingService(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void sendMailMessage() {
        this.mailSender.send(new MimeMessagePreparator() {

            @Override
            public void prepare(MimeMessage mimeMessage) throws Exception {
                MimeMessageHelper helper =
                    new MimeMessageHelper(mimeMessage, true, "UTF-8");
                helper.addTo("foo@bar.com");
                helper.setFrom("bar@baz.com");
                helper.addAttachment("test.txt", ...);
                helper.setSubject("test subject with attachment");
                helper.setText("mime body", false);
            }
        });
    }
}
```

10.4. Configuring regions

Amazon SES is not available in all regions of the Amazon Web Services cloud. Therefore an application hosted and operated in a region that does not support the mail service will produce an error while using the mail service. Therefore the region must be overridden for the mail sender configuration. The example below shows a typical combination of a region (EU-CENTRAL-1) that does not provide an SES service where the client is overridden to use a valid region (EU-WEST-1).

```
<beans ...>

 <aws-context:context-region region="eu-central-1" />
 <aws-mail:mail-sender id="testSender" region="eu-west-1"/>

</beans>
```

10.5. Authenticating e-mails

To avoid any spam attacks on the Amazon SES mail service, applications without production access must verify each e-mail receiver otherwise the mail sender will throw a

`com.amazonaws.services.simpleemail.model.MessageRejectedException`.

Production access can be requested and will disable the need for mail address verification.

Chapter 11. Resource handling

The Spring Framework provides a `org.springframework.core.io.ResourceLoader` abstraction to load files from the filesystem, servlet context and the classpath. Spring Cloud AWS adds support for the Amazon S3 service to load and write resources with the resource loader and the `s3` protocol.

The resource loader is part of the context module, therefore no additional dependencies are necessary to use the resource handling support.

11.1. Configuring the resource loader

Spring Cloud AWS does not modify the default resource loader unless it encounters an explicit configuration with an XML namespace element. The configuration consists of one element for the whole application context that is shown below:

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aws-context="http://www.springframework.org/schema/cloud/aws/context"
       xsi:schemaLocation="http://www.springframework.org/schema/cloud/aws/context
       http://www.springframework.org/schema/cloud/aws/context/spring-cloud-aws-
context.xsd">

    <aws-context:context-credentials>
            ...
        </aws-context:context-credentials>

    <aws-context:context-resource-loader/>
</beans>
```

11.2. Downloading files

Downloading files can be done by using the `s3` protocol to reference Amazon S3 buckets and objects inside their bucket. The typical pattern is `s3://<bucket>/<object>` where bucket is the global and unique bucket name and object is a valid object name inside the bucket. The object name can be a file in the *root* folder of a bucket or a nested file within a directory inside a bucket.

The next example demonstrates the use of the resource loader to load different resources.

```
public class SimpleResourceLoadingBean {

    @Autowired
    private ResourceLoader resourceLoader;

    public void resourceLoadingMethod() throws IOException {
        Resource resource =
this.resourceLoader.getResource("s3://myBucket/rootFile.log");
        Resource secondResource =
this.resourceLoader.getResource("s3://myBucket/rootFolder/subFile");

        InputStream inputStream = resource.getInputStream();
        //read file
    }
}
```

11.3. Uploading files

Since Spring Framework 3.1 the resource loader can also be used to upload files with the `org.springframework.core.io.WritableResource` interface which is a specialization of the `org.springframework.core.io.ResourceLoader` interface. Clients can upload files using the `WritableResource` interface. The next example demonstrates an upload of a resource using the resource loader.

```
public class SimpleResourceLoadingBean {

    @Autowired
    private ResourceLoader resourceLoader;

    public void writeResource() throws IOException {
        Resource resource =
this.resourceLoader.getResource("s3://myBucket/rootFile.log");
        WritableResource writableResource = (WritableResource) resource;
        try (OutputStream outputStream = writableResource.getOutputStream()) {
            outputStream.write("test".getBytes());
        }
    }
}
```

11.3.1. Uploading multi-part files

Amazon S3 supports multi-part uploads to increase the general throughput while uploading. Spring Cloud AWS by default only uses one thread to upload the files and therefore does not provide parallel upload support. Users can configure a custom `org.springframework.core.task.TaskExecutor` for the resource loader. The resource loader will queue multiple threads at the same time to use parallel multi-part uploads.

The configuration for a resource loader that uploads with 10 Threads looks like the following

```
<beans ...>
  <aws-context:context-resource-loader task-executor="executor" />
  <task:executor id="executor" pool-size="10" queue-capacity="0" rejection-
policy="CALLER_RUNS" />
</beans>
```

Spring Cloud AWS consumes up to 5 MB (at a minimum) of memory per thread. Therefore each parallel thread will incur a memory footprint of 5 MB in the heap, and a thread size of 10 will consume therefore up to 50 mb of heap space. Spring Cloud AWS releases the memory as soon as possible. Also, the example above shows that there is no `queue-capacity` configured, because queued requests would also consume memory.

11.3.2. Uploading with the TransferManager

The Amazon SDK also provides a high-level abstraction that is useful to upload files, also with multiple threads using the multi-part functionality. A `com.amazonaws.services.s3.transfer.TransferManager` can be easily created in the application code and injected with the pre-configured `com.amazonaws.services.s3.AmazonS3` client that is already created with the Spring Cloud AWS resource loader configuration.

This example shows the use of the `transferManager` within an application to upload files from the hard-drive.

```
public class SimpleResourceLoadingBean {

    @Autowired
    private AmazonS3 amazonS3;

    public void withTransferManager() {
        TransferManager transferManager = new TransferManager(this.amazonS3);
        transferManager.upload("myBucket","filename",new File("someFile"));
    }
}
```

11.4. Searching resources

The Spring resource loader also supports collecting resources based on an Ant-style path specification. Spring Cloud AWS offers the same support to resolve resources within a bucket and even throughout buckets. The actual resource loader needs to be wrapped with the Spring Cloud AWS one in order to search for s3 buckets, in case of non s3 bucket the resource loader will fall back to the original one. The next example shows the resource resolution by using different patterns.

```
public class SimpleResourceLoadingBean {

    private ResourcePatternResolver resourcePatternResolver;

    @Autowired
    public void setupResolver(ApplicationContext applicationContext, AmazonS3
amazonS3){
        this.resourcePatternResolver = new
PathMatchingSimpleStorageResourcePatternResolver(amazonS3, applicationContext);
    }

    public void resolveAndLoad() throws IOException {
        Resource[] allTxtFilesInFolder =
this.resourcePatternResolver.getResources("s3://bucket/name/*.txt");
        Resource[] allTxtFilesInBucket =
this.resourcePatternResolver.getResources("s3://bucket/**/*.txt");
        Resource[] allTxtFilesGlobally =
this.resourcePatternResolver.getResources("s3://**/*.txt");
    }
}
```

Resolving resources throughout all buckets can be very time consuming depending on the number of buckets a user owns.

11.5. Using CloudFormation

CloudFormation also allows to create buckets during stack creation. These buckets will typically have a generated name that must be used as the bucket name. In order to allow application developers to define *static* names inside their configuration, Spring Cloud AWS provides support to resolve the generated bucket names. Application developers can use the `org.springframework.cloud.aws.core.env.ResourceIdResolver` interface to resolve the physical names that are generated based on the logical names.

The next example shows a bucket definition inside a CloudFormation stack template. The bucket will be created with a name like *integrationteststack-sampleBucket-23qysofs62tc2*

```
{
    "Resources": {
        "sampleBucket": {
            "Type": "AWS::S3::Bucket"
        }
    }
}
```

Application developers can resolve that name and use it to load resources as shown in the next example below.

```
public class SimpleResourceLoadingBean {

    private final ResourceLoader loader;
    private final ResourceIdResolver idResolver;

    @Autowired
    public SimpleResourceLoadingBean(ResourceLoader loader, ResourceIdResolver
idResolver) {
        this.loader = loader;
        this.idResolver = idResolver;
    }

    public void resolveAndLoad() {
        String sampleBucketName = this.idResolver.
            resolveToPhysicalResourceId("sampleBucket");
        Resource resource = this.loader.
            getResource("s3://" + sampleBucketName + "/test");
    }
}
```

Chapter 12. CloudWatch Metrics

Spring Cloud AWS provides Spring Boot auto-configuration for Micrometer CloudWatch integration. To send metrics to CloudWatch add a dependency to `spring-cloud-aws-actuator` module:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-aws-actuator</artifactId>
</dependency>
```

Additionally CloudWatch integration requires a value provided for `management.metrics.export.cloudwatch.namespace` configuration property.

Following configuration properties are available to configure CloudWatch integration:

property	default	description
management.metrics.export.cloudwatch.namespace		The namespace which will be used when sending metrics to CloudWatch. This property is needed and must not be null.
management.metrics.export.cloudwatch.enabled	true	If CloudWatch integration should be enabled. This property should be likely set to `false` for a local development profile.
management.metrics.export.cloudwatch.step	1m	The interval at which metrics are sent to CloudWatch. The default is 1 minute.

Chapter 13. Configuration properties

To see the list of all Spring Cloud AWS related configuration properties please check the Appendix page.

Spring Cloud Build

[spring cloud build]

Spring Cloud Build is a common utility project for Spring Cloud to use for plugin and dependency management.

Chapter 14. Building and Deploying

To install locally:

```
$ mvn install -s .settings.xml
```

and to deploy snapshots to repo.spring.io:

```
$ mvn deploy
-DaltSnapshotDeploymentRepository=repo.spring.io::default::https://repo.spring.io/libs
-snapshot-local
```

for a RELEASE build use

```
$ mvn deploy
-DaltReleaseDeploymentRepository=repo.spring.io::default::https://repo.spring.io/libs
-release-local
```

and for jcenter use

```
$ mvn deploy
-DaltReleaseDeploymentRepository=bintray::default::https://api.bintray.com/maven/sprin
g/jars/org.springframework.cloud:build
```

and for Maven Central use

```
$ mvn deploy -P central -DaltReleaseDeploymentRepository=sonatype-nexus
-staging::default::https://oss.sonatype.org/service/local/staging/deploy/maven2
```

(the "central" profile is available for all projects in Spring Cloud and it sets up the gpg jar signing, and the repository has to be specified separately for this project because it is a parent of the starter parent which users in turn have as their own parent).

Chapter 15. Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

15.1. Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the Contributor License Agreement. Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

15.2. Code of Conduct

This project adheres to the Contributor Covenant code of conduct. By participating, you are expected to uphold this code. Please report unacceptable behavior to spring-code-of-conduct@pivotal.io.

15.3. Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the Spring Cloud Build project. If using IntelliJ, you can use the Eclipse Code Formatter Plugin to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the .java files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).
- When writing a commit message please follow these conventions, if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).

15.4. Checkstyle

Spring Cloud Build comes with a set of checkstyle rules. You can find them in the `spring-cloud-build-tools` module. The most notable files under the module are:

spring-cloud-build-tools/

```
└── src
    ├── checkstyle
    │   └── checkstyle-suppressions.xml ③
    └── main
        └── resources
            ├── checkstyle-header.txt ②
            └── checkstyle.xml ①
```

① Default Checkstyle rules

② File header setup

③ Default suppression rules

15.4.1. Checkstyle configuration

Checkstyle rules are **disabled by default**. To add checkstyle to your project just define the following properties and plugins.

pom.xml

```
<properties>
<maven-checkstyle-plugin.failsOnError>true</maven-checkstyle-plugin.failsOnError> ①
        <maven-checkstyle-plugin.failsOnViolation>true
        </maven-checkstyle-plugin.failsOnViolation> ②
        <maven-checkstyle-plugin.includeTestSourceDirectory>true
        </maven-checkstyle-plugin.includeTestSourceDirectory> ③
</properties>

<build>
        <plugins>
            <plugin> ④
                <groupId>io.spring.javaformat</groupId>
                <artifactId>spring-javaformat-maven-plugin</artifactId>
            </plugin>
            <plugin> ⑤
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
            </plugin>
        </plugins>

    <reporting>
        <plugins>
            <plugin> ⑤
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
            </plugin>
        </plugins>
    </reporting>
</build>
```

① Fails the build upon Checkstyle errors

② Fails the build upon Checkstyle violations

③ Checkstyle analyzes also the test sources

④ Add the Spring Java Format plugin that will reformat your code to pass most of the Checkstyle formatting rules

⑤ Add checkstyle plugin to your build and reporting phases

If you need to suppress some rules (e.g. line length needs to be longer), then it’s enough for you to define a file under `${project.root}/src/checkstyle/checkstyle-suppressions.xml` with your suppressions. Example:

projectRoot/src/checkstyle/checkstyle-suppresions.xml

```
<?xml version="1.0"?>
<!DOCTYPE suppressions PUBLIC
        "-//Puppy Crawl//DTD Suppressions 1.1//EN"
        "https://www.puppycrawl.com/dtds/suppressions_1_1.dtd">
<suppressions>
    <suppress files=".*ConfigServerApplication\.java"
checks="HideUtilityClassConstructor"/>
    <suppress files=".*ConfigClientWatch\.java" checks="LineLengthCheck"/>
</suppressions>
```

It's advisable to copy the `${spring-cloud-build.rootFolder}/.editorconfig` and `${spring-cloud-build.rootFolder}/.springformat` to your project. That way, some default formatting rules will be applied. You can do so by running this script:

```
$ curl https://raw.githubusercontent.com/spring-cloud/spring-cloud-
build/master/.editorconfig -o .editorconfig
$ touch .springformat
```

15.5. IDE setup

15.5.1. Intellij IDEA

In order to setup Intellij you should import our coding conventions, inspection profiles and set up the checkstyle plugin. The following files can be found in the Spring Cloud Build project.

spring-cloud-build-tools/

```
└── src
    ├── checkstyle
    │   └── checkstyle-suppressions.xml ③
    └── main
        └── resources
            ├── checkstyle-header.txt ②
            ├── checkstyle.xml ①
            └── intellij
                ├── Intellij_Project_Defaults.xml ④
                └── Intellij_Spring_Boot_Java_Conventions.xml ⑤
```

① Default Checkstyle rules

② File header setup

③ Default suppression rules

④ Project defaults for Intellij that apply most of Checkstyle rules

⑤ Project style conventions for Intellij that apply most of Checkstyle rules

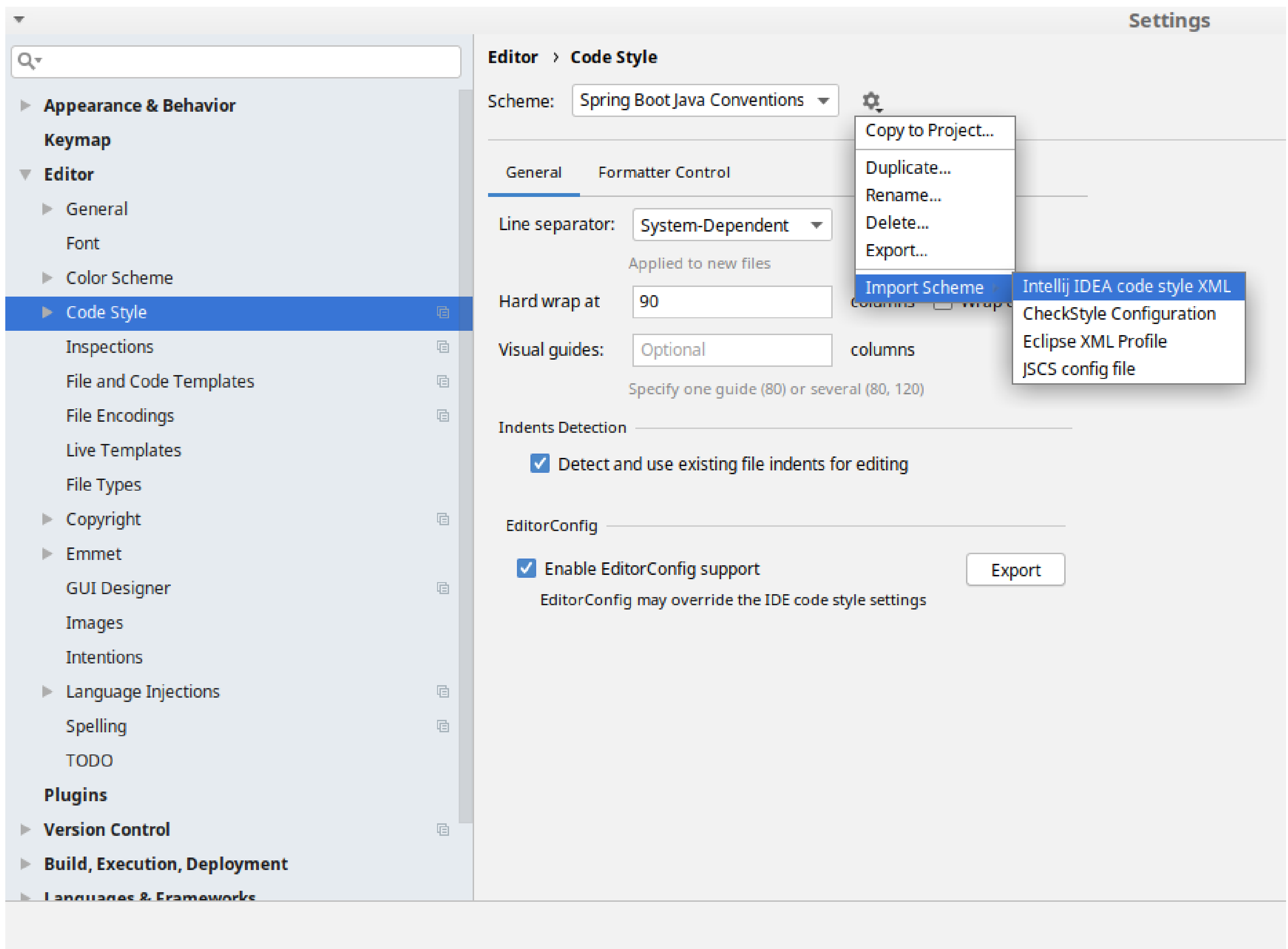


Figure 1. Code style

Go to `File` → `Settings` → `Editor` → `Code style`. There click on the icon next to the `Scheme` section. There, click on the `Import Scheme` value and pick the `Intellij IDEA code style XML` option. Import the `spring-cloud-build-tools/src/main/resources/intellij/Intellij_Spring_Boot_Java_Conventions.xml` file.

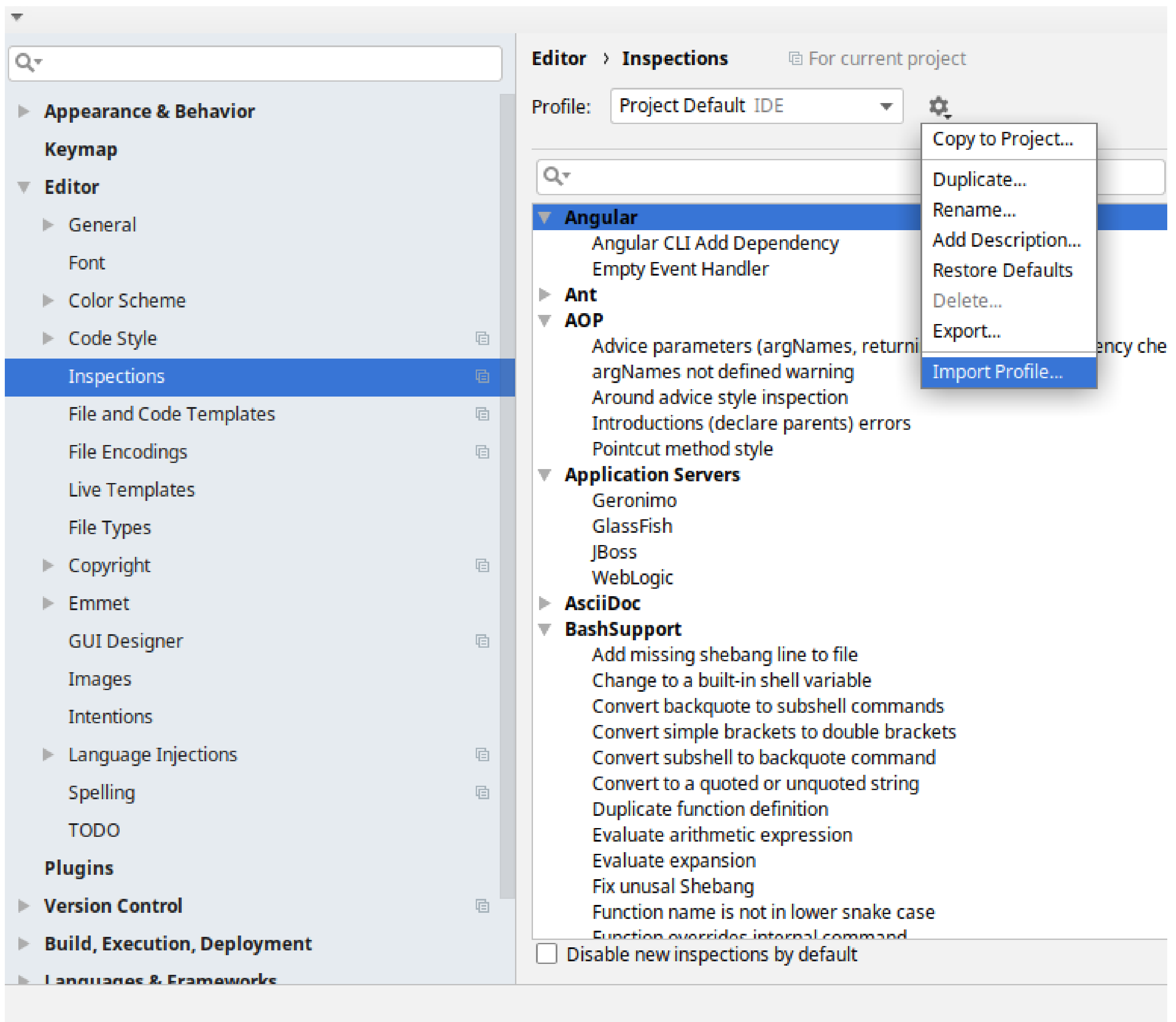
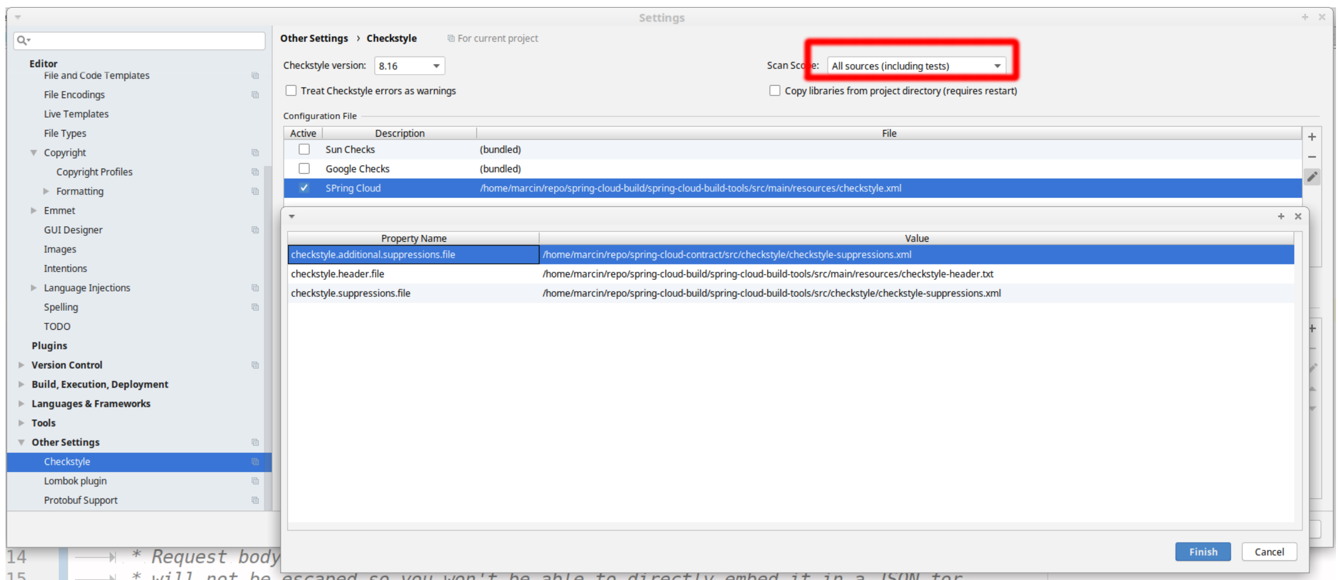


Figure 2. Inspection profiles

Go to `File` → `Settings` → `Editor` → `Inspections`. There click on the icon next to the `Profile` section. There, click on the `Import Profile` and import the `spring-cloud-build-tools/src/main/resources/intellij/Intellij_Project_Defaults.xml` file.

Checkstyle

To have Intellij work with Checkstyle, you have to install the `Checkstyle` plugin. It's advisable to also install the `Assertions2Assertj` to automatically convert the JUnit assertions



Go to `File` → `Settings` → `Other settings` → `Checkstyle`. There click on the `+` icon in the `Configuration file` section. There, you'll have to define where the checkstyle rules should be picked from. In the image above, we've picked the rules from the cloned Spring Cloud Build repository. However, you can point to the Spring Cloud Build's GitHub repository (e.g. for the `checkstyle.xml` : `raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle.xml`). We need to provide the following variables:

- `checkstyle.header.file` - please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/main/resources/checkstyle-header.txt` file either in your cloned repo or via the `raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle-header.txt` URL.
- `checkstyle.suppressions.file` - default suppressions. Please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml` file either in your cloned repo or via the `raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml` URL.
- `checkstyle.additional.suppressions.file` - this variable corresponds to suppressions in your local project. E.g. you're working on `spring-cloud-contract`. Then point to the `project-root/src/checkstyle/checkstyle-suppressions.xml` folder. Example for `spring-cloud-contract` would be: `/home/username/spring-cloud-contract/src/checkstyle/checkstyle-suppressions.xml`.

Remember to set the `Scan Scope` to `All sources` since we apply checkstyle rules for production and test sources.

Chapter 16. Flattening the POMs

To avoid propagating build setup that is required to build a Spring Cloud project, we're using the maven flatten plugin. It has the advantage of letting you use whatever features you need while publishing "clean" pom to the repository.

In order to add it, add the `org.codehaus.mojo:flatten-maven-plugin` to your `pom.xml`.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>flatten-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Chapter 17. Reusing the documentation

Spring Cloud Build publishes its `spring-cloud-build-docs` module that contains helpful scripts (e.g. README generation ruby script) and css, xslt and images for the Spring Cloud documentation. If you want to follow the same convention approach of generating documentation just add these plugins to your `docs` module

```
<profiles>
        <profile>
            <id>docs</id>
            <build>
                <plugins>
                    <plugin>
                        <groupId>pl.project13.maven</groupId>
                        <artifactId>git-commit-id-plugin</artifactId> ①
                    </plugin>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-dependency-plugin</artifactId> ②
                    </plugin>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-resources-plugin</artifactId> ③
                    </plugin>
                    <plugin>
                        <groupId>org.codehaus.mojo</groupId>
                        <artifactId>exec-maven-plugin</artifactId> ④
                    </plugin>
                    <plugin>
                        <groupId>org.asciidoctor</groupId>
                        <artifactId>asciidoctor-maven-plugin</artifactId> ⑤
                    </plugin>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-antrun-plugin</artifactId> ⑥
                    </plugin>
                </plugins>
            </build>
        </profile>
    </profiles>
```

① This plugin downloads sets up all the git information of the project

② This plugin downloads the resources of the `spring-cloud-build-docs` module

③ This plugin unpacks the resources of the `spring-cloud-build-docs` module

④ This plugin generates an `adoc` file with all the configuration properties from the classpath

⑤ This plugin is required to parse the Asciidoctor documentation

⑥ This plugin is required to copy resources into proper final destinations and to generate main

README.adoc and to assert that no files use unresolved links

The order of plugin declaration is important!

In order for the build to generate the `adoc` file with all your configuration properties, your `docs` module should contain all the dependencies on the classpath, that you would want to scan for configuration properties. The file will be output to `${docsModule}/src/main/asciidoc/_configprops.adoc` file (configurable via the `configprops.path` property).

If you want to modify which of the configuration properties are put in the table, you can tweak the `configprops.inclusionPattern` pattern to include only a subset of the properties (e.g. `<configprops.inclusionPattern>spring.sleuth.*</configprops.inclusionPattern>`).

Spring Cloud Build Docs comes with a set of attributes for asciidoctor that you can reuse.

```
<attributes>
    <docinfo>shared</docinfo>
    <allow-uri-read>true</allow-uri-read>
    <nofooter/>
    <toc>left</toc>
    <toc-levels>4</toc-levels>
    <sectlinks>true</sectlinks>
    <sources-root>${project.basedir}/src@</sources-root>
    <asciidoc-sources-root>${project.basedir}/src/main/asciidoc@</asciidoc-sources-
root>
    <generated-resources-root>${project.basedir}/target/generated-resources@
    </generated-resources-root>
    <!-- Use this attribute the reference code from another module -->
    <!-- Note the @ at the end, lowering the precedence of the attribute -->
    <project-root>${maven.multiModuleProjectDirectory}@</project-root>
    <!-- It's mandatory for you to pass the docs.main property -->
    <github-repo>${docs.main}@</github-repo>
    <github-project>https://github.com/spring-cloud/${docs.main}@</github-project>
    <github-raw>
        https://raw.githubusercontent.com/spring-cloud/${docs.main}/${github-tag}@
    </github-raw>
    <github-code>https://github.com/spring-cloud/${docs.main}/tree/${github-tag}@
    </github-code>
    <github-issues>https://github.com/spring-cloud/${docs.main}/issues/@</github-
issues>
    <github-wiki>https://github.com/spring-cloud/${docs.main}/wiki@</github-wiki>
    <github-master-code>https://github.com/spring-cloud/${docs.main}/tree/master@
    </github-master-code>
    <index-link>${index-link}@</index-link>

    <!-- Spring Cloud specific -->
    <!-- for backward compatibility -->
    <spring-cloud-version>${project.version}@</spring-cloud-version>
    <project-version>${project.version}@</project-version>
    <github-tag>${github-tag}@</github-tag>
    <version-type>${version-type}@</version-type>
    <docs-url>https://docs.spring.io/${docs.main}/docs/${project.version}@</docs-url>
    <raw-docs-url>${github-raw}@</raw-docs-url>
    <project-version>${project.version}@</project-version>
    <project-name>${docs.main}@</project-name>
</attributes>
```

Chapter 18. Updating the guides

We assume that your project contains guides under the `guides` folder.

```
.
└── guides
    ├── gs-guide1
    ├── gs-guide2
    └── gs-guide3
```

This means that the project contains 3 guides that would correspond to the following guides in Spring Guides org.

- github.com/spring-guides/gs-guide1
- github.com/spring-guides/gs-guide2
- github.com/spring-guides/gs-guide3

If you deploy your project with the `-Pguides` profile like this

```
$ ./mvnw clean deploy -Pguides
```

what will happen is that for GA project versions, we will clone `gs-guide1`, `gs-guide2` and `gs-guide3` and update their contents with the ones being under your `guides` project.

You can skip this by either not adding the `guides` profile, or passing the `-DskipGuides` system property when the profile is turned on.

You can configure the project version passed to guides via the `guides-project.version` (defaults to `${project.version}`). The phase at which guides get updated can be configured by `guides-update.phase` (defaults to `deploy`).

Spring Cloud Bus

Spring Cloud Bus links the nodes of a distributed system with a lightweight message broker. This broker can then be used to broadcast state changes (such as configuration changes) or other management instructions. A key idea is that the bus is like a distributed actuator for a Spring Boot application that is scaled out. However, it can also be used as a communication channel between apps. This project provides starters for either an AMQP broker or Kafka as the transport.

Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at github.

Chapter 19. Quick Start

Spring Cloud Bus works by adding Spring Boot autconfiguration if it detects itself on the classpath. To enable the bus, add `spring-cloud-starter-bus-amqp` or `spring-cloud-starter-bus-kafka` to your dependency management. Spring Cloud takes care of the rest. Make sure the broker (RabbitMQ or Kafka) is available and configured. When running on localhost, you need not do anything. If you run remotely, use Spring Cloud Connectors or Spring Boot conventions to define the broker credentials, as shown in the following example for Rabbit:

application.yml

```
spring:
  rabbitmq:
    host: mybroker.com
    port: 5672
    username: user
    password: secret
```

The bus currently supports sending messages to all nodes listening or all nodes for a particular service (as defined by Eureka). The `/bus/*` actuator namespace has some HTTP endpoints. Currently, two are implemented. The first, `/bus/env`, sends key/value pairs to update each node's Spring Environment. The second, `/bus/refresh`, reloads each application's configuration, as though they had all been pinged on their `/refresh` endpoint.

The Spring Cloud Bus starters cover Rabbit and Kafka, because those are the two most common implementations. However, Spring Cloud Stream is quite flexible, and the binder works with `spring-cloud-bus`.

Chapter 20. Bus Endpoints

Spring Cloud Bus provides two endpoints, `/actuator/bus-refresh` and `/actuator/bus-env` that correspond to individual actuator endpoints in Spring Cloud Commons, `/actuator/refresh` and `/actuator/env` respectively.

20.1. Bus Refresh Endpoint

The `/actuator/bus-refresh` endpoint clears the `RefreshScope` cache and rebinds `@ConfigurationProperties`. See the Refresh Scope documentation for more information.

To expose the `/actuator/bus-refresh` endpoint, you need to add following configuration to your application:

```
management.endpoints.web.exposure.include=bus-refresh
```

20.2. Bus Env Endpoint

The `/actuator/bus-env` endpoint updates each instances environment with the specified key/value pair across multiple instances.

To expose the `/actuator/bus-env` endpoint, you need to add following configuration to your application:

```
management.endpoints.web.exposure.include=bus-env
```

The `/actuator/bus-env` endpoint accepts `POST` requests with the following shape:

```
{
    "name": "key1",
    "value": "value1"
}
```

Chapter 21. Addressing an Instance

Each instance of the application has a service ID, whose value can be set with `spring.cloud.bus.id` and whose value is expected to be a colon-separated list of identifiers, in order from least specific to most specific. The default value is constructed from the environment as a combination of the `spring.application.name` and `server.port` (or `spring.application.index`, if set). The default value of the ID is constructed in the form of `app:index:id`, where:

- `app` is the `vcap.application.name`, if it exists, or `spring.application.name`
- `index` is the `vcap.application.instance_index`, if it exists, `spring.application.index`, `local.server.port`, `server.port`, or `0` (in that order).
- `id` is the `vcap.application.instance_id`, if it exists, or a random value.

The HTTP endpoints accept a “destination” path parameter, such as `/bus-refresh/customers:9000`, where `destination` is a service ID. If the ID is owned by an instance on the bus, it processes the message, and all other instances ignore it.

Chapter 22. Addressing All Instances of a Service

The “destination” parameter is used in a Spring `PathMatcher` (with the path separator as a colon — `:`) to determine if an instance processes the message. Using the example from earlier, `/bus-env/customers:**` targets all instances of the “customers” service regardless of the rest of the service ID.

Chapter 23. Service ID Must Be Unique

The bus tries twice to eliminate processing an event — once from the original `ApplicationEvent` and once from the queue. To do so, it checks the sending service ID against the current service ID. If multiple instances of a service have the same ID, events are not processed. When running on a local machine, each service is on a different port, and that port is part of the ID. Cloud Foundry supplies an index to differentiate. To ensure that the ID is unique outside Cloud Foundry, set `spring.application.index` to something unique for each instance of a service.

Chapter 24. Customizing the Message Broker

Spring Cloud Bus uses Spring Cloud Stream to broadcast the messages. So, to get messages to flow, you need only include the binder implementation of your choice in the classpath. There are convenient starters for the bus with AMQP (RabbitMQ) and Kafka (`spring-cloud-starter-bus-[amqp|kafka]`). Generally speaking, Spring Cloud Stream relies on Spring Boot autoconfiguration conventions for configuring middleware. For instance, the AMQP broker address can be changed with `spring.rabbitmq.*` configuration properties. Spring Cloud Bus has a handful of native configuration properties in `spring.cloud.bus.*` (for example, `spring.cloud.bus.destination` is the name of the topic to use as the external middleware). Normally, the defaults suffice.

To learn more about how to customize the message broker settings, consult the Spring Cloud Stream documentation.

Chapter 25. Tracing Bus Events

Bus events (subclasses of `RemoteApplicationEvent`) can be traced by setting `spring.cloud.bus.trace.enabled=true`. If you do so, the Spring Boot `TraceRepository` (if it is present) shows each event sent and all the acks from each service instance. The following example comes from the `/trace` endpoint:

```
{
  "timestamp": "2015-11-26T10:24:44.411+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "stores:8081",
    "destination": "*:**"
  }
  },
  {
  "timestamp": "2015-11-26T10:24:41.864+0000",
  "info": {
    "signal": "spring.cloud.bus.sent",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
  },
  {
  "timestamp": "2015-11-26T10:24:41.862+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
}
```

The preceding trace shows that a `RefreshRemoteApplicationEvent` was sent from `customers:9000`, broadcast to all services, and received (acked) by `customers:9000` and `stores:8081`.

To handle the ack signals yourself, you could add an `@EventListener` for the `AckRemoteApplicationEvent` and `SentApplicationEvent` types to your app (and enable tracing). Alternatively, you could tap into the `TraceRepository` and mine the data from there.

Any Bus application can trace acks. However, sometimes, it is useful to do this in a central service that can do more complex queries on the data or forward it to a specialized tracing service.

Chapter 26. Broadcasting Your Own Events

The Bus can carry any event of type `RemoteApplicationEvent`. The default transport is JSON, and the deserializer needs to know which types are going to be used ahead of time. To register a new type, you must put it in a subpackage of `org.springframework.cloud.bus.event`.

To customise the event name, you can use `@JsonTypeName` on your custom class or rely on the default strategy, which is to use the simple name of the class.

Both the producer and the consumer need access to the class definition.

26.1. Registering events in custom packages

If you cannot or do not want to use a subpackage of `org.springframework.cloud.bus.event` for your custom events, you must specify which packages to scan for events of type `RemoteApplicationEvent` by using the `@RemoteApplicationEventScan` annotation. Packages specified with `@RemoteApplicationEventScan` include subpackages.

For example, consider the following custom event, called `MyEvent`:

```
package com.acme;

public class MyEvent extends RemoteApplicationEvent {
    ...
}
```

You can register that event with the deserializer in the following way:

```
package com.acme;

@Configuration
@RemoteApplicationEventScan
public class BusConfiguration {
    ...
}
```

Without specifying a value, the package of the class where `@RemoteApplicationEventScan` is used is registered. In this example, `com.acme` is registered by using the package of `BusConfiguration`.

You can also explicitly specify the packages to scan by using the `value`, `basePackages` or `basePackageClasses` properties on `@RemoteApplicationEventScan`, as shown in the following example:

```
package com.acme;

@Configuration
//@RemoteApplicationEventScan({"com.acme", "foo.bar"})
//@RemoteApplicationEventScan(basePackages = {"com.acme", "foo.bar", "fizz.buzz"})
@RemoteApplicationEventScan(basePackageClasses = BusConfiguration.class)
public class BusConfiguration {
    ...
}
```

All of the preceding examples of `@RemoteApplicationEventScan` are equivalent, in that the `com.acme` package is registered by explicitly specifying the packages on `@RemoteApplicationEventScan`.

You can specify multiple base packages to scan.

Chapter 27. Configuration properties

To see the list of all Bus related configuration properties please check the Appendix page.

Spring Cloud Circuit Breaker

Hoxton.SR8

.1. Configuring Resilience4J Circuit Breakers

.1.1. Starters

There are two starters for the Resilience4J implementations, one for reactive applications and one for non-reactive applications.

- `org.springframework.cloud:spring-cloud-starter-circuitbreaker-resilience4j` - non-reactive applications
- `org.springframework.cloud:spring-cloud-starter-circuitbreaker-reactor-resilience4j` - reactive applications

.1.2. Auto-Configuration

You can disable the Resilience4J auto-configuration by setting `spring.cloud.circuitbreaker.resilience4j.enabled` to `false`.

.1.3. Default Configuration

To provide a default configuration for all of your circuit breakers create a `Customize` bean that is passed a `Resilience4JCircuitBreakerFactory` or `ReactiveResilience4JCircuitBreakerFactory`. The `configureDefault` method can be used to provide a default configuration.

```
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new
Resilience4JConfigBuilder(id)

.timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4
)).build())
            .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())
            .build());
}
```

Reactive Example

```
@Bean
public Customizer<ReactiveResilience4JCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new
Resilience4JConfigBuilder(id)
            .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())

.timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4
)).build()).build());
}
```

.1.4. Specific Circuit Breaker Configuration

Similarly to providing a default configuration, you can create a `Customize` bean this is passed a `Resilience4JCircuitBreakerFactory` or `ReactiveResilience4JCircuitBreakerFactory`.

```
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> slowCustomizer() {
    return factory -> factory.configure(builder ->
builder.circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())

.timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2
)).build()), "slow");
}
```

In addition to configuring the circuit breaker that is created you can also customize the circuit breaker after it has been created but before it is returned to the caller. To do this you can use the `addCircuitBreakerCustomizer` method. This can be useful for adding event handlers to Resilience4J circuit breakers.

```
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> slowCustomizer() {
    return factory -> factory.addCircuitBreakerCustomizer(circuitBreaker ->
circuitBreaker.getEventPublisher()
    .onError(normalFluxErrorConsumer).onSuccess(normalFluxSuccessConsumer),
"normalflux");
}
```

Reactive Example

```
@Bean
public Customizer<ReactiveResilience4JCircuitBreakerFactory> slowCusomtizer() {
    return factory -> {
        factory.configure(builder -> builder

.timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2
)).build())
        .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults()), "slow",
"slowflux");
        factory.addCircuitBreakerCustomizer(circuitBreaker ->
circuitBreaker.getEventPublisher()

.onError(normalFluxErrorConsumer).onSuccess(normalFluxSuccessConsumer),
"normalflux");
     };
}
```

.1.5. Collecting Metrics

Spring Cloud Circuit Breaker Resilience4j includes auto-configuration to setup metrics collection as long as the right dependencies are on the classpath. To enable metric collection you must include `org.springframework.boot:spring-boot-starter-actuator`, and `io.github.resilience4j:resilience4j-micrometer`. For more information on the metrics that get produced when these dependencies are present, see the Resilience4j documentation.

You don't have to include `micrometer-core` directly as it is brought in by `spring-boot-starter-actuator`

.2. Configuring Spring Retry Circuit Breakers

Spring Retry provides declarative retry support for Spring applications. A subset of the project includes the ability to implement circuit breaker functionality. Spring Retry provides a circuit breaker implementation via a combination of it's `CircuitBreakerRetryPolicy` and a stateful retry. All circuit breakers created using Spring Retry will be created using the `CircuitBreakerRetryPolicy` and a `DefaultRetryState`. Both of these classes can be configured using `SpringRetryConfigBuilder`.

.2.1. Default Configuration

To provide a default configuration for all of your circuit breakers create a `Customize` bean that is passed a `SpringRetryCircuitBreakerFactory`. The `configureDefault` method can be used to provide a default configuration.

```
@Bean
public Customizer<SpringRetryCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new
SpringRetryConfigBuilder(id)
        .retryPolicy(new TimeoutRetryPolicy()).build());
}
```

.2.2. Specific Circuit Breaker Configuration

Similarly to providing a default configuration, you can create a `Customize` bean this is passed a `SpringRetryCircuitBreakerFactory`.

```
@Bean
public Customizer<SpringRetryCircuitBreakerFactory> slowCustomizer() {
    return factory -> factory.configure(builder -> builder.retryPolicy(new
SimpleRetryPolicy(1)).build(), "slow");
}
```

In addition to configuring the circuit breaker that is created you can also customize the circuit breaker after it has been created but before it is returned to the caller. To do this you can use the `addRetryTemplateCustomizers` method. This can be useful for adding event handlers to the `RetryTemplate`.

```
@Bean
public Customizer<SpringRetryCircuitBreakerFactory> slowCustomizer() {
    return factory -> factory.addRetryTemplateCustomizers(retryTemplate ->
retryTemplate.registerListener(new RetryListener() {

        @Override
        public <T, E extends Throwable> boolean open(RetryContext context,
RetryCallback<T, E> callback) {
            return false;
        }

        @Override
        public <T, E extends Throwable> void close(RetryContext context,
RetryCallback<T, E> callback, Throwable throwable) {

        }

        @Override
        public <T, E extends Throwable> void onError(RetryContext context,
RetryCallback<T, E> callback, Throwable throwable) {

        }
    }));
}
```

Chapter 28. Building

28.1. Basic Compile and Test

To build the source you will need to install JDK 1.8.

Spring Cloud uses Maven for most build-related activities, and you should be able to get off the ground quite quickly by cloning the project you are interested in and typing

```
$ ./mvnw install
```

You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.

Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

For hints on how to build the project look in `.travis.yml` if there is one. There should be a "script" and maybe "install" command. Also look at the "services" section to see if any services need to be running locally (e.g. mongo or rabbit). Ignore the git-related bits that you might find in "before_install" since they're related to setting git credentials and you already have those.

The projects that require middleware generally include a `docker-compose.yml`, so consider using Docker Compose to run the middeware servers in Docker containers. See the README in the scripts demo repository for specific instructions about the common cases of mongo, rabbit and redis.

If all else fails, build with the command from `.travis.yml` (usually `./mvnw install`).

28.2. Documentation

The spring-cloud-build module has a "docs" profile, and if you switch that on it will try to build asciidoc sources from `src/main/asciidoc`. As part of that process it will look for a `README.adoc` and process it by loading all the includes, but not parsing or rendering it, just copying it to `${main.basedir}` (defaults to `$/tmp/releaser-1598648275631-0/spring-cloud-release/train-docs/target/unpacked-docs`, i.e. the root of the project). If there are any changes in the README it will then show up after a Maven build as a modified file in the correct place. Just commit it and push the change.

28.3. Working with the code

If you don't have an IDE preference we would recommend that you use Spring Tools Suite or Eclipse when working with the code. We use the m2eclipse eclipse plugin for maven support. Other IDEs and tools should also work without issue as long as they use Maven 3.3.3 or better.

28.3.1. Activate the Spring Maven profile

Spring Cloud projects require the 'spring' Maven profile to be activated to resolve the spring milestone and snapshot repositories. Use your preferred IDE to set this profile to be active, or you may experience build errors.

28.3.2. Importing into eclipse with m2eclipse

We recommend the m2eclipse eclipse plugin when working with eclipse. If you don't already have m2eclipse installed it is available from the "eclipse marketplace".

Older versions of m2e do not support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the right profile for the projects. If you see many different errors related to the POMs in the projects, check that you have an up to date installation. If you can't upgrade m2e, add the "spring" profile to your `settings.xml`. Alternatively you can copy the repository settings from the "spring" profile of the parent pom into your `settings.xml`.

28.3.3. Importing into eclipse without m2eclipse

If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu.

Chapter 29. Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

29.1. Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the Contributor License Agreement. Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

29.2. Code of Conduct

This project adheres to the Contributor Covenant code of conduct. By participating, you are expected to uphold this code. Please report unacceptable behavior to spring-code-of-conduct@pivotal.io.

29.3. Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the Spring Cloud Build project. If using IntelliJ, you can use the Eclipse Code Formatter Plugin to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the .java files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).
- When writing a commit message please follow these conventions, if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).

29.4. Checkstyle

Spring Cloud Build comes with a set of checkstyle rules. You can find them in the `spring-cloud-build-tools` module. The most notable files under the module are:

spring-cloud-build-tools/

```
└── src
    ├── checkstyle
    │   └── checkstyle-suppressions.xml ③
    └── main
        └── resources
            ├── checkstyle-header.txt ②
            └── checkstyle.xml ①
```

① Default Checkstyle rules

② File header setup

③ Default suppression rules

29.4.1. Checkstyle configuration

Checkstyle rules are **disabled by default**. To add checkstyle to your project just define the following properties and plugins.

pom.xml

```
<properties>
<maven-checkstyle-plugin.failsOnError>true</maven-checkstyle-plugin.failsOnError> ①
        <maven-checkstyle-plugin.failsOnViolation>true
        </maven-checkstyle-plugin.failsOnViolation> ②
        <maven-checkstyle-plugin.includeTestSourceDirectory>true
        </maven-checkstyle-plugin.includeTestSourceDirectory> ③
</properties>

<build>
        <plugins>
            <plugin> ④
                <groupId>io.spring.javaformat</groupId>
                <artifactId>spring-javaformat-maven-plugin</artifactId>
            </plugin>
            <plugin> ⑤
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
            </plugin>
        </plugins>

    <reporting>
        <plugins>
            <plugin> ⑤
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
            </plugin>
        </plugins>
    </reporting>
</build>
```

① Fails the build upon Checkstyle errors

② Fails the build upon Checkstyle violations

③ Checkstyle analyzes also the test sources

④ Add the Spring Java Format plugin that will reformat your code to pass most of the Checkstyle formatting rules

⑤ Add checkstyle plugin to your build and reporting phases

If you need to suppress some rules (e.g. line length needs to be longer), then it's enough for you to define a file under `${project.root}/src/checkstyle/checkstyle-suppressions.xml` with your suppressions. Example:

projectRoot/src/checkstyle/checkstyle-suppresions.xml

```
<?xml version="1.0"?>
<!DOCTYPE suppressions PUBLIC
        "-//Puppy Crawl//DTD Suppressions 1.1//EN"
        "https://www.puppycrawl.com/dtds/suppressions_1_1.dtd">
<suppressions>
    <suppress files=".*ConfigServerApplication\.java"
checks="HideUtilityClassConstructor"/>
    <suppress files=".*ConfigClientWatch\.java" checks="LineLengthCheck"/>
</suppressions>
```

It's advisable to copy the `${spring-cloud-build.rootFolder}/.editorconfig` and `${spring-cloud-build.rootFolder}/.springformat` to your project. That way, some default formatting rules will be applied. You can do so by running this script:

```
$ curl https://raw.githubusercontent.com/spring-cloud/spring-cloud-
build/master/.editorconfig -o .editorconfig
$ touch .springformat
```

29.5. IDE setup

29.5.1. Intellij IDEA

In order to setup Intellij you should import our coding conventions, inspection profiles and set up the checkstyle plugin. The following files can be found in the Spring Cloud Build project.

spring-cloud-build-tools/

```
└── src
    ├── checkstyle
    │   └── checkstyle-suppressions.xml ③
    └── main
        └── resources
            ├── checkstyle-header.txt ②
            ├── checkstyle.xml ①
            └── intellij
                ├── Intellij_Project_Defaults.xml ④
                └── Intellij_Spring_Boot_Java_Conventions.xml ⑤
```

① Default Checkstyle rules

② File header setup

③ Default suppression rules

④ Project defaults for Intellij that apply most of Checkstyle rules

⑤ Project style conventions for Intellij that apply most of Checkstyle rules

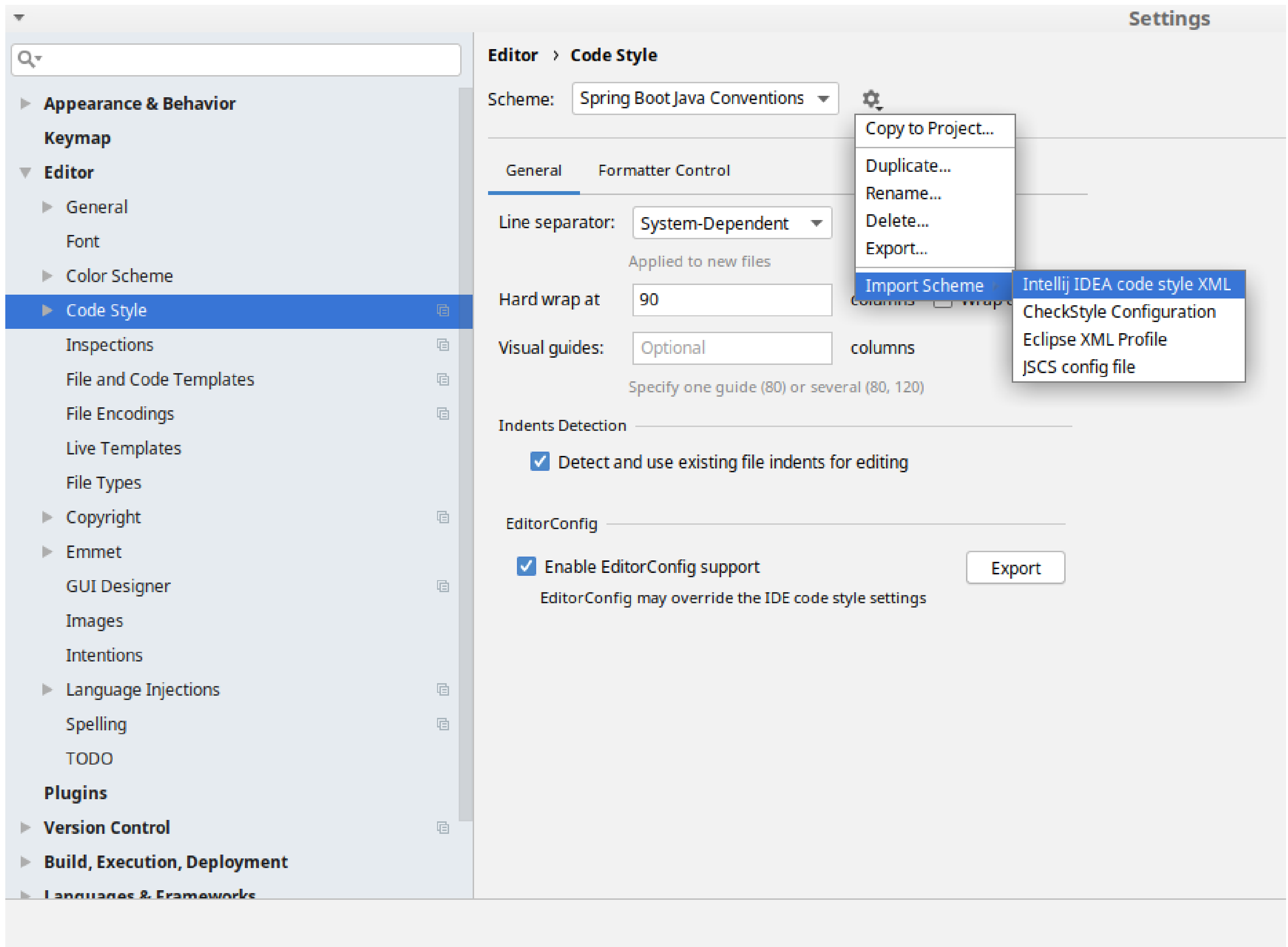


Figure 3. Code style

Go to `File` → `Settings` → `Editor` → `Code style`. There click on the icon next to the `Scheme` section. There, click on the `Import Scheme` value and pick the `Intellij IDEA code style XML` option. Import the `spring-cloud-build-tools/src/main/resources/intellij/Intellij_Spring_Boot_Java_Conventions.xml` file.

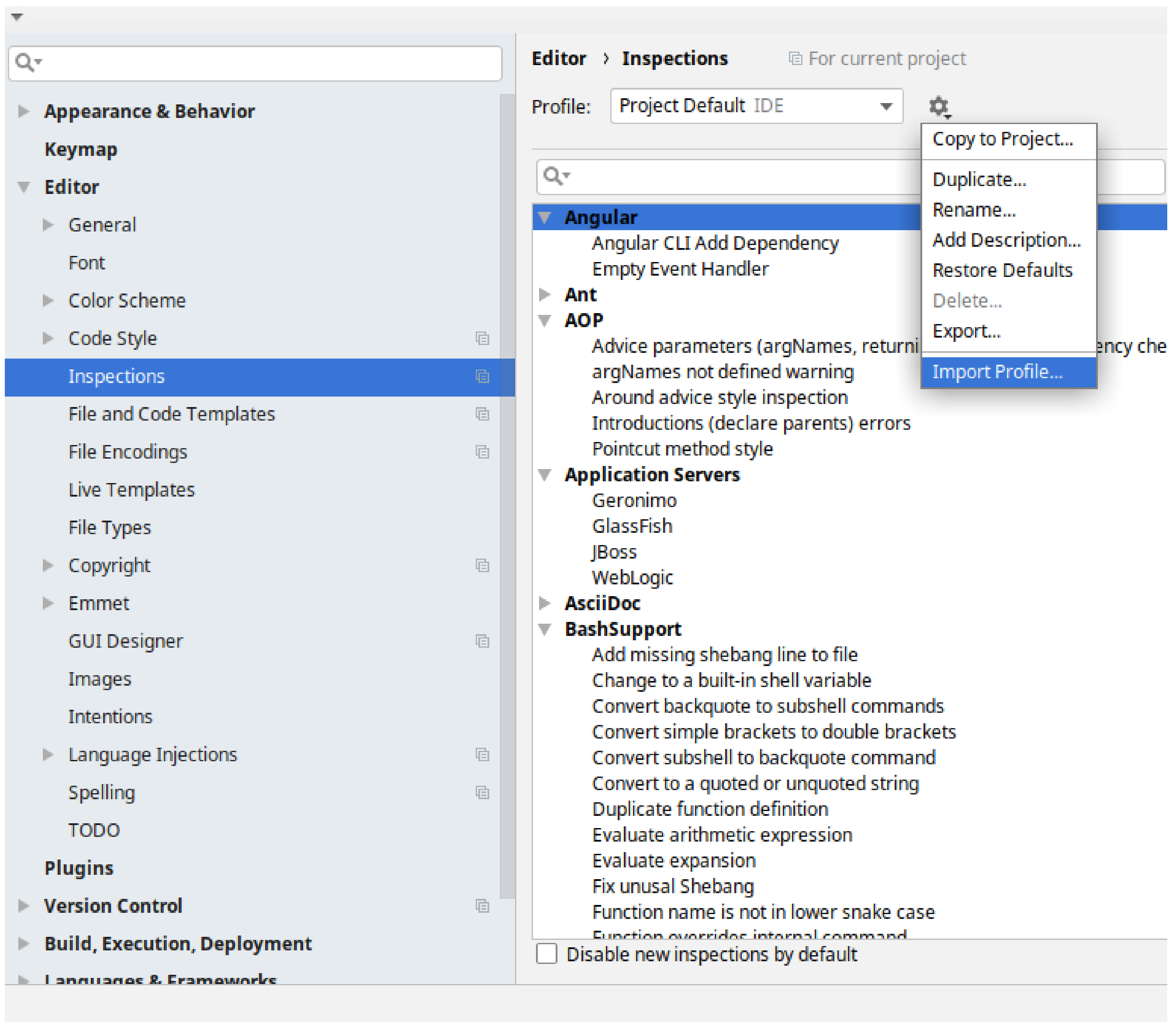
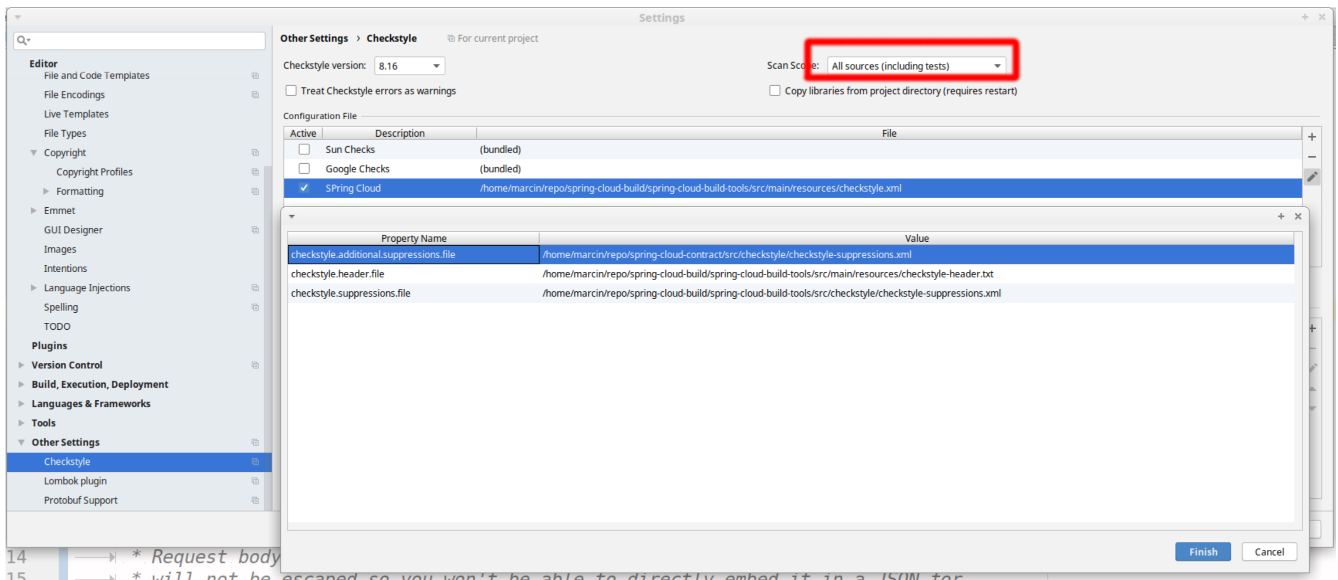


Figure 4. Inspection profiles

Go to `File` → `Settings` → `Editor` → `Inspections`. There click on the icon next to the `Profile` section. There, click on the `Import Profile` and import the `spring-cloud-build-tools/src/main/resources/intellij/Intellij_Project_Defaults.xml` file.

Checkstyle

To have Intellij work with Checkstyle, you have to install the `Checkstyle` plugin. It's advisable to also install the `Assertions2Assertj` to automatically convert the JUnit assertions



Go to `File` → `Settings` → `Other settings` → `Checkstyle`. There click on the `+` icon in the `Configuration file` section. There, you'll have to define where the checkstyle rules should be picked from. In the image above, we've picked the rules from the cloned Spring Cloud Build repository. However, you can point to the Spring Cloud Build's GitHub repository (e.g. for the `checkstyle.xml` : raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle.xml). We need to provide the following variables:

- `checkstyle.header.file` - please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/main/resources/checkstyle-header.txt` file either in your cloned repo or via the raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle-header.txt URL.
- `checkstyle.suppressions.file` - default suppressions. Please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml` file either in your cloned repo or via the raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml URL.
- `checkstyle.additional.suppressions.file` - this variable corresponds to suppressions in your local project. E.g. you're working on `spring-cloud-contract`. Then point to the `project-root/src/checkstyle/checkstyle-suppressions.xml` folder. Example for `spring-cloud-contract` would be: `/home/username/spring-cloud-contract/src/checkstyle/checkstyle-suppressions.xml`.

Remember to set the `Scan Scope` to `All sources` since we apply checkstyle rules for production and test sources.

Spring Boot Cloud CLI

Spring Boot CLI provides Spring Boot command line features for Spring Cloud. You can write Groovy scripts to run Spring Cloud component applications (e.g. `@EnableEurekaServer`). You can also easily do things like encryption and decryption to support Spring Cloud Config clients with secret configuration values. With the Launcher CLI you can launch services like Eureka, Zipkin, Config Server conveniently all at once from the command line (very useful at development time).

Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at github.

Chapter 30. Installation

To install, make sure you have Spring Boot CLI (2.0.0 or better):

```
$ spring version
Spring CLI v2.2.3.RELEASE
```

E.g. for SDKMan users

```
$ sdk install springboot 2.2.3.RELEASE
$ sdk use springboot 2.2.3.RELEASE
```

and install the Spring Cloud plugin

```
$ mvn install
$ spring install org.springframework.cloud:spring-cloud-cli:2.2.0.RELEASE
```

Prerequisites: to use the encryption and decryption features you need the full-strength JCE installed in your JVM (it's not there by default). You can download the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files" from Oracle, and follow instructions for installation (essentially replace the 2 policy files in the JRE lib/security directory with the ones that you downloaded).

Chapter 31. Running Spring Cloud Services in Development

The Launcher CLI can be used to run common services like Eureka, Config Server etc. from the command line. To list the available services you can do `spring cloud --list`, and to launch a default set of services just `spring cloud`. To choose the services to deploy, just list them on the command line, e.g.

```
$ spring cloud eureka configserver h2 kafka stubrunner zipkin
```

Summary of supported deployables:

Service	Name	Address	Description
eureka	Eureka Server	localhost:8761	Eureka server for service registration and discovery. All the other services show up in its catalog by default.
configserver	Config Server	localhost:8888	Spring Cloud Config Server running in the "native" profile and serving configuration from the local directory ./launcher
h2	H2 Database	localhost:9095 (console), jdbc:h2:tcp://localhost:9096/{data}	Relation database service. Use a file path for `{data}` (e.g. `./target/test`) when you connect. Remember that you can add `;MODE=MYSQL` or `;MODE=POSTGRESQL` to connect with compatibility to other server types.
kafka	Kafka Broker	localhost:9091 (actuator endpoints), localhost:9092	

Service	Name	Address	Description
hystrixdashboard	Hystrix Dashboard	localhost:7979	Any Spring Cloud app that declares Hystrix circuit breakers publishes metrics on `/hystrix.stream`. Type that address into the dashboard to visualize all the metrics,
dataflow	Dataflow Server	localhost:9393	Spring Cloud Dataflow server with UI at /admin-ui. Connect the Dataflow shell to target at root path.
zipkin	Zipkin Server	localhost:9411	Zipkin Server with UI for visualizing traces. Stores span data in memory and accepts them via HTTP POST of JSON data.
stubrunner	Stub Runner Boot	localhost:8750	Downloads WireMock stubs, starts WireMock and feeds the started servers with stored stubs. Pass `stubrunner.ids` to pass stub coordinates and then go to `localhost:8750/stubs`.

Each of these apps can be configured using a local YAML file with the same name (in the current working directory or a subdirectory called "config" or in `~/.spring-cloud`). E.g. in `configserver.yml` you might want to do something like this to locate a local git repository for the backend:

configserver.yml

```
spring:
  profiles:
    active: git
  cloud:
    config:
      server:
        git:
          uri: file://${user.home}/dev/demo/config-repo
```

E.g. in Stub Runner app you could fetch stubs from your local `.m2` in the following way.

stubrunner.yml

```
stubrunner:
  workOffline: true
  ids:
    - com.example:beer-api-producer:+:9876
```

31.1. Adding Additional Applications

Additional applications can be added to `./config/cloud.yml` (not `./config.yml` because that would replace the defaults), e.g. with

config/cloud.yml

```
spring:
  cloud:
    launcher:
      deployables:
        source:
          coordinates: maven://com.example:source:0.0.1-SNAPSHOT
          port: 7000
        sink:
          coordinates: maven://com.example:sink:0.0.1-SNAPSHOT
          port: 7001
```

when you list the apps:

```
$ spring cloud --list
source sink configserver dataflow eureka h2 hystrixdashboard kafka stubrunner zipkin
```

(notice the additional apps at the start of the list).

Chapter 32. Writing Groovy Scripts and Running Applications

Spring Cloud CLI has support for most of the Spring Cloud declarative features, such as the `@Enable*` class of annotations. For example, here is a fully functional Eureka server

app.groovy

```
@EnableEurekaServer
class Eureka {}
```

which you can run from the command line like this

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

To include additional dependencies, often it suffices just to add the appropriate feature-enabling annotation, e.g. `@EnableConfigServer`, `@EnableOAuth2Sso` or `@EnableEurekaClient`. To manually include a dependency you can use a `@Grab` with the special "Spring Boot" short style artifact co-ordinates, i.e. with just the artifact ID (no need for group or version information), e.g. to set up a client app to listen on AMQP for management events from the Spring CLoud Bus:

app.groovy

```
@Grab('spring-cloud-starter-bus-amqp')
@RestController
class Service {
  @RequestMapping('/')
  def home() { [message: 'Hello'] }
}
```

Chapter 33. Encryption and Decryption

The Spring Cloud CLI comes with an "encrypt" and a "decrypt" command. Both accept arguments in the same form with a key specified as a mandatory "--key", e.g.

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

To use a key in a file (e.g. an RSA public key for encyption) prepend the key value with "@" and provide the file path, e.g.

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

Spring Cloud for Cloud Foundry

Spring Cloud for Cloudfoundry makes it easy to run Spring Cloud apps in Cloud Foundry (the Platform as a Service). Cloud Foundry has the notion of a "service", which is middlware that you "bind" to an app, essentially providing it with an environment variable containing credentials (e.g. the location and username to use for the service).

The `spring-cloud-cloudfoundry-commons` module configures the Reactor-based Cloud Foundry Java client, v 3.0, and can be used standalone.

The `spring-cloud-cloudfoundry-web` project provides basic support for some enhanced features of webapps in Cloud Foundry: binding automatically to single-sign-on services and optionally enabling sticky routing for discovery.

The `spring-cloud-cloudfoundry-discovery` project provides an implementation of Spring Cloud Commons `DiscoveryClient` so you can `@EnableDiscoveryClient` and provide your credentials as `spring.cloud.cloudfoundry.discovery.[username,password]` (also `*.url` if you are not connecting to Pivotal Web Services) and then you can use the `DiscoveryClient` directly or via a `LoadBalancerClient`.

The first time you use it the discovery client might be slow owing to the fact that it has to get an access token from Cloud Foundry.

Chapter 34. Discovery

Here's a Spring Cloud app with Cloud Foundry discovery:

app.groovy

```
@Grab('org.springframework.cloud:spring-cloud-cloudfoundry')
@RestController
@EnableDiscoveryClient
class Application {

  @Autowired
  DiscoveryClient client

  @RequestMapping('/')
  String home() {
    'Hello from ' + client.getLocalServiceInstance()
  }

}
```

If you run it without any service bindings:

```
$ spring jar app.jar app.groovy
$ cf push -p app.jar
```

It will show its app name in the home page.

The `DiscoveryClient` can lists all the apps in a space, according to the credentials it is authenticated with, where the space defaults to the one the client is running in (if any). If neither org nor space are configured, they default per the user's profile in Cloud Foundry.

Chapter 35. Single Sign On

All of the OAuth2 SSO and resource server features moved to Spring Boot in version 1.3. You can find documentation in the Spring Boot user guide.

This project provides automatic binding from CloudFoundry service credentials to the Spring Boot features. If you have a CloudFoundry service called "sso", for instance, with credentials containing "client_id", "client_secret" and "auth_domain", it will bind automatically to the Spring OAuth2 client that you enable with `@EnableOAuth2Sso` (from Spring Boot). The name of the service can be parameterized using `spring.oauth2.sso.serviceId`.

Chapter 36. Configuration

To see the list of all Spring Cloud Sloud Foundry related configuration properties please check the Appendix page.

Cloud Native Applications

Cloud Native is a style of application development that encourages easy adoption of best practices in the areas of continuous delivery and value-driven development. A related discipline is that of building 12-factor Applications, in which development practices are aligned with delivery and operations goals — for instance, by using declarative programming and management and monitoring. Spring Cloud facilitates these styles of development in a number of specific ways. The starting point is a set of features to which all components in a distributed system need easy access.

Many of those features are covered by Spring Boot, on which Spring Cloud builds. Some more features are delivered by Spring Cloud as two libraries: Spring Cloud Context and Spring Cloud Commons. Spring Cloud Context provides utilities and special services for the `ApplicationContext` of a Spring Cloud application (bootstrap context, encryption, refresh scope, and environment endpoints). Spring Cloud Commons is a set of abstractions and common classes used in different Spring Cloud implementations (such as Spring Cloud Netflix and Spring Cloud Consul).

If you get an exception due to "Illegal key size" and you use Sun's JDK, you need to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. See the following links for more information:

- Java 6 JCE
- Java 7 JCE
- Java 8 JCE

Extract the files into the JDK/jre/lib/security folder for whichever version of JRE/JDK x64/x86 you use.

Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, you can find the source code and issue trackers for the project at github.

Chapter 37. Spring Cloud Context: Application Context Services

Spring Boot has an opinionated view of how to build an application with Spring. For instance, it has conventional locations for common configuration files and has endpoints for common management and monitoring tasks. Spring Cloud builds on top of that and adds a few features that many components in a system would use or occasionally need.

37.1. The Bootstrap Application Context

A Spring Cloud application operates by creating a “bootstrap” context, which is a parent context for the main application. This context is responsible for loading configuration properties from the external sources and for decrypting properties in the local external configuration files. The two contexts share an `Environment`, which is the source of external properties for any Spring application. By default, bootstrap properties (not `bootstrap.properties` but properties that are loaded during the bootstrap phase) are added with high precedence, so they cannot be overridden by local configuration.

The bootstrap context uses a different convention for locating external configuration than the main application context. Instead of `application.yml` (or `.properties`), you can use `bootstrap.yml`, keeping the external configuration for bootstrap and main context nicely separate. The following listing shows an example:

Example 1. bootstrap.yml

```
spring:
  application:
    name: foo
  cloud:
    config:
      uri: ${SPRING_CONFIG_URI:http://localhost:8888}
```

If your application needs any application-specific configuration from the server, it is a good idea to set the `spring.application.name` (in `bootstrap.yml` or `application.yml`). For the property `spring.application.name` to be used as the application’s context ID, you must set it in `bootstrap.[properties | yml]`.

If you want to retrieve specific profile configuration, you should also set `spring.profiles.active` in `bootstrap.[properties | yml]`.

You can disable the bootstrap process completely by setting `spring.cloud.bootstrap.enabled=false` (for example, in system properties).

37.2. Application Context Hierarchies

If you build an application context from `SpringApplication` or `SpringApplicationBuilder`, the Bootstrap context is added as a parent to that context. It is a feature of Spring that child contexts inherit property sources and profiles from their parent, so the “main” application context contains additional property sources, compared to building the same context without Spring Cloud Config. The additional property sources are:

- “bootstrap”: If any `PropertySourceLocators` are found in the bootstrap context and if they have non-empty properties, an optional `CompositePropertySource` appears with high priority. An example would be properties from the Spring Cloud Config Server. See “Customizing the Bootstrap Property Sources” for how to customize the contents of this property source.
- “applicationConfig: [classpath:bootstrap.yml]” (and related files if Spring profiles are active): If you have a `bootstrap.yml` (or `.properties`), those properties are used to configure the bootstrap context. Then they get added to the child context when its parent is set. They have lower precedence than the `application.yml` (or `.properties`) and any other property sources that are added to the child as a normal part of the process of creating a Spring Boot application. See “Changing the Location of Bootstrap Properties” for how to customize the contents of these property sources.

Because of the ordering rules of property sources, the “bootstrap” entries take precedence. However, note that these do not contain any data from `bootstrap.yml`, which has very low precedence but can be used to set defaults.

You can extend the context hierarchy by setting the parent context of any `ApplicationContext` you create — for example, by using its own interface or with the `SpringApplicationBuilder` convenience methods (`parent()`, `child()` and `sibling()`). The bootstrap context is the parent of the most senior ancestor that you create yourself. Every context in the hierarchy has its own “bootstrap” (possibly empty) property source to avoid promoting values inadvertently from parents down to their descendants. If there is a config server, every context in the hierarchy can also (in principle) have a different `spring.application.name` and, hence, a different remote property source. Normal Spring application context behavior rules apply to property resolution: properties from a child context override those in the parent, by name and also by property source name. (If the child has a property source with the same name as the parent, the value from the parent is not included in the child).

Note that the `SpringApplicationBuilder` lets you share an `Environment` amongst the whole hierarchy, but that is not the default. Thus, sibling contexts (in particular) do not need to have the same profiles or property sources, even though they may share common values with their parent.

37.3. Changing the Location of Bootstrap Properties

The `bootstrap.yml` (or `.properties`) location can be specified by setting `spring.cloud.bootstrap.name` (default: `bootstrap`), `spring.cloud.bootstrap.location` (default: empty) or `spring.cloud.bootstrap.additional-location` (default: empty) — for example, in System properties.

Those properties behave like the `spring.config.*` variants with the same name. With `spring.cloud.bootstrap.location` the default locations are replaced and only the specified ones are

used. To add locations to the list of default ones, `spring.cloud.bootstrap.additional-location` could be used. In fact, they are used to set up the bootstrap `ApplicationContext` by setting those properties in its `Environment`. If there is an active profile (from `spring.profiles.active` or through the `Environment` API in the context you are building), properties in that profile get loaded as well, the same as in a regular Spring Boot app — for example, from `bootstrap-development.properties` for a `development` profile.

37.4. Overriding the Values of Remote Properties

The property sources that are added to your application by the bootstrap context are often "remote" (from example, from Spring Cloud Config Server). By default, they cannot be overridden locally. If you want to let your applications override the remote properties with their own system properties or config files, the remote property source has to grant it permission by setting `spring.cloud.config.allowOverride=true` (it does not work to set this locally). Once that flag is set, two finer-grained settings control the location of the remote properties in relation to system properties and the application's local configuration:

- `spring.cloud.config.overrideNone=true`: Override from any local property source.
- `spring.cloud.config.overrideSystemProperties=false`: Only system properties, command line arguments, and environment variables (but not the local config files) should override the remote settings.

37.5. Customizing the Bootstrap Configuration

The bootstrap context can be set to do anything you like by adding entries to `/META-INF/spring.factories` under a key named `org.springframework.cloud.bootstrap.BootstrapConfiguration`. This holds a comma-separated list of Spring `@Configuration` classes that are used to create the context. Any beans that you want to be available to the main application context for autowiring can be created here. There is a special contract for `@Beans` of type `ApplicationContextInitializer`. If you want to control the startup sequence, you can mark classes with the `@Order` annotation (the default order is `last`).

When adding custom `BootstrapConfiguration`, be careful that the classes you add are not `@ComponentScanned` by mistake into your "main" application context, where they might not be needed. Use a separate package name for boot configuration classes and make sure that name is not already covered by your `@ComponentScan` or `@SpringBootApplication` annotated configuration classes.

The bootstrap process ends by injecting initializers into the main `SpringApplication` instance (which is the normal Spring Boot startup sequence, whether it runs as a standalone application or is deployed in an application server). First, a bootstrap context is created from the classes found in `spring.factories`. Then, all `@Beans` of type `ApplicationContextInitializer` are added to the main `SpringApplication` before it is started.

37.6. Customizing the Bootstrap Property Sources

The default property source for external configuration added by the bootstrap process is the Spring Cloud Config Server, but you can add additional sources by adding beans of type `PropertySourceLocator` to the bootstrap context (through `spring.factories`). For instance, you can insert additional properties from a different server or from a database.

As an example, consider the following custom locator:

```
@Configuration
public class CustomPropertySourceLocator implements PropertySourceLocator {

    @Override
    public PropertySource<?> locate(Environment environment) {
        return new MapPropertySource("customProperty",
                Collections.<String,
Object>singletonMap("property.from.sample.custom.source", "worked as intended"));
    }

}
```

The `Environment` that is passed in is the one for the `ApplicationContext` about to be created — in other words, the one for which we supply additional property sources. It already has its normal Spring Boot-provided property sources, so you can use those to locate a property source specific to this `Environment` (for example, by keying it on `spring.application.name`, as is done in the default Spring Cloud Config Server property source locator).

If you create a jar with this class in it and then add a `META-INF/spring.factories` containing the following setting, the `customProperty` `PropertySource` appears in any application that includes that jar on its classpath:

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=sample.custom.CustomPro
pertySourceLocator
```

37.7. Logging Configuration

If you use Spring Boot to configure log settings, you should place this configuration in `bootstrap.[yml | properties]` if you would like it to apply to all events.

For Spring Cloud to initialize logging configuration properly, you cannot use a custom prefix. For example, using `custom.loggin.logpath` is not recognized by Spring Cloud when initializing the logging system.

37.8. Environment Changes

The application listens for an EnvironmentChangeEvent and reacts to the change in a couple of standard ways (additional ApplicationListeners can be added as @Beans in the normal way). When an EnvironmentChangeEvent is observed, it has a list of key values that have changed, and the application uses those to:

- Re-bind any @ConfigurationProperties beans in the context.
- Set the logger levels for any properties in logging.level.*.

Note that the Spring Cloud Config Client does not, by default, poll for changes in the Environment. Generally, we would not recommend that approach for detecting changes (although you could set it up with a @Scheduled annotation). If you have a scaled-out client application, it is better to broadcast the EnvironmentChangeEvent to all the instances instead of having them polling for changes (for example, by using the Spring Cloud Bus).

The EnvironmentChangeEvent covers a large class of refresh use cases, as long as you can actually make a change to the Environment and publish the event. Note that those APIs are public and part of core Spring). You can verify that the changes are bound to @ConfigurationProperties beans by visiting the /configprops endpoint (a standard Spring Boot Actuator feature). For instance, a DataSource can have its maxPoolSize changed at runtime (the default DataSource created by Spring Boot is a @ConfigurationProperties bean) and grow capacity dynamically. Re-binding @ConfigurationProperties does not cover another large class of use cases, where you need more control over the refresh and where you need a change to be atomic over the whole ApplicationContext. To address those concerns, we have @RefreshScope.

37.9. Refresh Scope

When there is a configuration change, a Spring @Bean that is marked as @RefreshScope gets special treatment. This feature addresses the problem of stateful beans that get their configuration injected only when they are initialized. For instance, if a DataSource has open connections when the database URL is changed through the Environment, you probably want the holders of those connections to be able to complete what they are doing. Then, the next time something borrows a connection from the pool, it gets one with the new URL.

Sometimes, it might even be mandatory to apply the @RefreshScope annotation on some beans that can be only initialized once. If a bean is "immutable", you have to either annotate the bean with @RefreshScope or specify the classname under the property key: spring.cloud.refresh.extra-refreshable.

If you hava a DataSource bean that is a HikariDataSource, it can not be refreshed. It is the default value for spring.cloud.refresh.never-refreshable. Choose a different DataSource implementation if you need it to be refreshed.

Refresh scope beans are lazy proxies that initialize when they are used (that is, when a method is called), and the scope acts as a cache of initialized values. To force a bean to re-initialize on the next method call, you must invalidate its cache entry.

The `RefreshScope` is a bean in the context and has a public `refreshAll()` method to refresh all beans in the scope by clearing the target cache. The `/refresh` endpoint exposes this functionality (over HTTP or JMX). To refresh an individual bean by name, there is also a `refresh(String)` method.

To expose the `/refresh` endpoint, you need to add following configuration to your application:

```
management:
  endpoints:
    web:
      exposure:
        include: refresh
```

`@RefreshScope` works (technically) on a `@Configuration` class, but it might lead to surprising behavior. For example, it does not mean that all the `@Beans` defined in that class are themselves in `@RefreshScope`. Specifically, anything that depends on those beans cannot rely on them being updated when a refresh is initiated, unless it is itself in `@RefreshScope`. In that case, it is rebuilt on a refresh and its dependencies are re-injected. At that point, they are re-initialized from the refreshed `@Configuration`).

37.10. Encryption and Decryption

Spring Cloud has an `Environment` pre-processor for decrypting property values locally. It follows the same rules as the Spring Cloud Config Server and has the same external configuration through `encrypt.*`. Thus, you can use encrypted values in the form of `{cipher}*`, and, as long as there is a valid key, they are decrypted before the main application context gets the `Environment` settings. To use the encryption features in an application, you need to include Spring Security RSA in your classpath (Maven co-ordinates: `org.springframework.security:spring-security-rsa`), and you also need the full strength JCE extensions in your JVM.

If you get an exception due to "Illegal key size" and you use Sun's JDK, you need to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. See the following links for more information:

- Java 6 JCE
- Java 7 JCE
- Java 8 JCE

Extract the files into the JDK/jre/lib/security folder for whichever version of JRE/JDK x64/x86 you use.

37.11. Endpoints

For a Spring Boot Actuator application, some additional management endpoints are available. You can use:

- `POST` to `/actuator/env` to update the `Environment` and rebind `@ConfigurationProperties` and log levels.
- `/actuator/refresh` to re-load the boot strap context and refresh the `@RefreshScope` beans.
- `/actuator/restart` to close the `ApplicationContext` and restart it (disabled by default).
- `/actuator/pause` and `/actuator/resume` for calling the `Lifecycle` methods (`stop()` and `start()` on the `ApplicationContext`).

If you disable the `/actuator/restart` endpoint then the `/actuator/pause` and `/actuator/resume` endpoints will also be disabled since they are just a special case of `/actuator/restart`.

Chapter 38. Spring Cloud Commons: Common Abstractions

Patterns such as service discovery, load balancing, and circuit breakers lend themselves to a common abstraction layer that can be consumed by all Spring Cloud clients, independent of the implementation (for example, discovery with Eureka or Consul).

38.1. The `@EnableDiscoveryClient` Annotation

Spring Cloud Commons provides the `@EnableDiscoveryClient` annotation. This looks for implementations of the `DiscoveryClient` and `ReactiveDiscoveryClient` interfaces with `META-INF/spring.factories`. Implementations of the discovery client add a configuration class to `spring.factories` under the `org.springframework.cloud.client.discovery.EnableDiscoveryClient` key. Examples of `DiscoveryClient` implementations include Spring Cloud Netflix Eureka, Spring Cloud Consul Discovery, and Spring Cloud Zookeeper Discovery.

Spring Cloud will provide both the blocking and reactive service discovery clients by default. You can disable the blocking and/or reactive clients easily by setting `spring.cloud.discovery.blocking.enabled=false` or `spring.cloud.discovery.reactive.enabled=false`. To completely disable service discovery you just need to set `spring.cloud.discovery.enabled=false`.

By default, implementations of `DiscoveryClient` auto-register the local Spring Boot server with the remote discovery server. This behavior can be disabled by setting `autoRegister=false` in `@EnableDiscoveryClient`.

`@EnableDiscoveryClient` is no longer required. You can put a `DiscoveryClient` implementation on the classpath to cause the Spring Boot application to register with the service discovery server.

38.1.1. Health Indicator

Commons creates a Spring Boot `HealthIndicator` that `DiscoveryClient` implementations can participate in by implementing `DiscoveryHealthIndicator`. To disable the composite `HealthIndicator`, set `spring.cloud.discovery.client.composite-indicator.enabled=false`. A generic `HealthIndicator` based on `DiscoveryClient` is auto-configured (`DiscoveryClientHealthIndicator`). To disable it, set `spring.cloud.discovery.client.health-indicator.enabled=false`. To disable the description field of the `DiscoveryClientHealthIndicator`, set `spring.cloud.discovery.client.health-indicator.include-description=false`. Otherwise, it can bubble up as the `description` of the rolled up `HealthIndicator`.

38.1.2. Ordering `DiscoveryClient` instances

`DiscoveryClient` interface extends `Ordered`. This is useful when using multiple discovery clients, as it allows you to define the order of the returned discovery clients, similar to how you can order the beans loaded by a Spring application. By default, the order of any `DiscoveryClient` is set to `0`. If you want to set a different order for your custom `DiscoveryClient` implementations, you just need to override the `getOrder()` method so that it returns the value that is suitable for your setup. Apart

from this, you can use properties to set the order of the `DiscoveryClient` implementations provided by Spring Cloud, among others `ConsulDiscoveryClient`, `EurekaDiscoveryClient` and `ZookeeperDiscoveryClient`. In order to do it, you just need to set the `spring.cloud.{clientIdentifier}.discovery.order` (or `eureka.client.order` for Eureka) property to the desired value.

38.1.3. SimpleDiscoveryClient

If there is no Service-Registry-backed `DiscoveryClient` in the classpath, `SimpleDiscoveryClient` instance, that uses properties to get information on service and instances, will be used.

The information about the available instances should be passed to via properties in the following format: `spring.cloud.discovery.client.simple.instances.service1[0].uri=http://s11:8080`, where `spring.cloud.discovery.client.simple.instances` is the common prefix, then `service1` stands for the ID of the service in question, while `[0]` indicates the index number of the instance (as visible in the example, indexes start with `0`), and then the value of `uri` is the actual URI under which the instance is available.

38.2. ServiceRegistry

Commons now provides a `ServiceRegistry` interface that provides methods such as `register(Registration)` and `deregister(Registration)`, which let you provide custom registered services. `Registration` is a marker interface.

The following example shows the `ServiceRegistry` in use:

```
@Configuration
@EnableDiscoveryClient(autoRegister=false)
public class MyConfiguration {
    private ServiceRegistry registry;

    public MyConfiguration(ServiceRegistry registry) {
        this.registry = registry;
    }

    // called through some external process, such as an event or a custom actuator
endpoint
    public void register() {
        Registration registration = constructRegistration();
        this.registry.register(registration);
    }
}
```

Each `ServiceRegistry` implementation has its own `Registry` implementation.

- `ZookeeperRegistration` used with `ZookeeperServiceRegistry`

- `EurekaRegistration` used with `EurekaServiceRegistry`
- `ConsulRegistration` used with `ConsulServiceRegistry`

If you are using the `ServiceRegistry` interface, you are going to need to pass the correct `Registry` implementation for the `ServiceRegistry` implementation you are using.

38.2.1. ServiceRegistry Auto-Registration

By default, the `ServiceRegistry` implementation auto-registers the running service. To disable that behavior, you can set: * `@EnableDiscoveryClient(autoRegister=false)` to permanently disable auto-registration. * `spring.cloud.service-registry.auto-registration.enabled=false` to disable the behavior through configuration.

ServiceRegistry Auto-Registration Events

There are two events that will be fired when a service auto-registers. The first event, called `InstancePreRegisteredEvent`, is fired before the service is registered. The second event, called `InstanceRegisteredEvent`, is fired after the service is registered. You can register an `ApplicationListener`(s) to listen to and react to these events.

These events will not be fired if the `spring.cloud.service-registry.auto-registration.enabled` property is set to `false`.

38.2.2. Service Registry Actuator Endpoint

Spring Cloud Commons provides a `/service-registry` actuator endpoint. This endpoint relies on a `Registration` bean in the Spring Application Context. Calling `/service-registry` with GET returns the status of the `Registration`. Using POST to the same endpoint with a JSON body changes the status of the current `Registration` to the new value. The JSON body has to include the `status` field with the preferred value. Please see the documentation of the `ServiceRegistry` implementation you use for the allowed values when updating the status and the values returned for the status. For instance, Eureka's supported statuses are `UP`, `DOWN`, `OUT_OF_SERVICE`, and `UNKNOWN`.

38.3. Spring RestTemplate as a Load Balancer Client

You can configure a `RestTemplate` to use a Load-balancer client. To create a load-balanced `RestTemplate`, create a `RestTemplate` `@Bean` and use the `@LoadBalanced` qualifier, as the following example shows:

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    public String doOtherStuff() {
        String results = restTemplate.getForObject("http://stores/stores",
String.class);
        return results;
    }
}
```

A `RestTemplate` bean is no longer created through auto-configuration. Individual applications must create it.

The URI needs to use a virtual host name (that is, a service name, not a host name). The Ribbon client is used to create a full physical address. See {githubroot}/spring-cloud-netflix/blob/master/spring-cloud-netflix-ribbon/src/main/java/org/springframework/cloud/netflix/ribbon/RibbonAutoConfiguration.java[`RibbonAutoConfiguration`] for the details of how the `RestTemplate` is set up.

To use a load-balanced `RestTemplate`, you need to have a load-balancer implementation in your classpath. The recommended implementation is `BlockingLoadBalancerClient`. Add Spring Cloud LoadBalancer starter to your project in order to use it. The `RibbonLoadBalancerClient` also can be used, but it's now under maintenance and we do not recommend adding it to new projects.

By default, if you have both `RibbonLoadBalancerClient` and `BlockingLoadBalancerClient`, to preserve backward compatibility, `RibbonLoadBalancerClient` is used. To override it, you can set the `spring.cloud.loadbalancer.ribbon.enabled` property to `false`.

38.4. Spring WebClient as a Load Balancer Client

You can configure `WebClient` to automatically use a load-balancer client. To create a load-balanced `WebClient`, create a `WebClient.Builder` `@Bean` and use the `@LoadBalanced` qualifier, as follows:

```
@Configuration
public class MyConfiguration {

    @Bean
    @LoadBalanced
    public WebClient.Builder loadBalancedWebClientBuilder() {
        return WebClient.builder();
    }
}

public class MyClass {
    @Autowired
    private WebClient.Builder webClientBuilder;

    public Mono<String> doOtherStuff() {
        return webClientBuilder.build().get().uri("http://stores/stores")
                        .retrieve().bodyToMono(String.class);
    }
}
```

The URI needs to use a virtual host name (that is, a service name, not a host name). The Ribbon client or Spring Cloud LoadBalancer is used to create a full physical address.

If you want to use a `@LoadBalanced WebClient.Builder`, you need to have a load balancer implementation in the classpath. We recommend that you add the Spring Cloud LoadBalancer starter to your project. Then, `ReactiveLoadBalancer` is used underneath. Alternatively, this functionality also works with `spring-cloud-starter-netflix-ribbon`, but the request is handled by a non-reactive `LoadBalancerClient` under the hood. Additionally, `spring-cloud-starter-netflix-ribbon` is already in maintenance mode, so we do not recommend adding it to new projects. If you have both `spring-cloud-starter-loadbalancer` and `spring-cloud-starter-netflix-ribbon` in your classpath, Ribbon is used by default. To switch to Spring Cloud LoadBalancer, set the `spring.cloud.loadbalancer.ribbon.enabled` property to `false`.

38.4.1. Retrying Failed Requests

A load-balanced `RestTemplate` can be configured to retry failed requests. By default, this logic is disabled. You can enable it by adding Spring Retry to your application's classpath. The load-balanced `RestTemplate` honors some of the Ribbon configuration values related to retrying failed requests. You can use `client.ribbon.MaxAutoRetries`, `client.ribbon.MaxAutoRetriesNextServer`, and `client.ribbon.OkToRetryOnAllOperations` properties. If you would like to disable the retry logic with Spring Retry on the classpath, you can set `spring.cloud.loadbalancer.retry.enabled=false`. See the Ribbon documentation for a description of what these properties do.

If you would like to implement a `BackOffPolicy` in your retries, you need to create a bean of type `LoadBalancedRetryFactory` and override the `createBackOffPolicy` method:

```
@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedRetryFactory retryFactory() {
        return new LoadBalancedRetryFactory() {
            @Override
            public BackOffPolicy createBackOffPolicy(String service) {
                return new ExponentialBackOffPolicy();
            }
        };
    }
}
```

`client` in the preceding examples should be replaced with your Ribbon client's name.

If you want to add one or more `RetryListener` implementations to your retry functionality, you need to create a bean of type `LoadBalancedRetryListenerFactory` and return the `RetryListener` array you would like to use for a given service, as the following example shows:

```
@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedRetryListenerFactory retryListenerFactory() {
        return new LoadBalancedRetryListenerFactory() {
            @Override
            public RetryListener[] createRetryListeners(String service) {
                return new RetryListener[]{new RetryListener() {
                    @Override
                    public <T, E extends Throwable> boolean open(RetryContext
context, RetryCallback<T, E> callback) {
                        //TODO Do you business...
                        return true;
                    }

                    @Override
                     public <T, E extends Throwable> void close(RetryContext
context, RetryCallback<T, E> callback, Throwable throwable) {
                        //TODO Do you business...
                    }

                    @Override
                    public <T, E extends Throwable> void onError(RetryContext
context, RetryCallback<T, E> callback, Throwable throwable) {
                        //TODO Do you business...
                    }
                }};
            }
        };
    }
}
```

38.5. Multiple `RestTemplate` Objects

If you want a `RestTemplate` that is not load-balanced, create a `RestTemplate` bean and inject it. To access the load-balanced `RestTemplate`, use the `@LoadBalanced` qualifier when you create your `@Bean`, as the following example shows:

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate loadBalanced() {
        return new RestTemplate();
    }

    @Primary
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

public class MyClass {
@Autowired
private RestTemplate restTemplate;

    @Autowired
    @LoadBalanced
    private RestTemplate loadBalanced;

    public String doOtherStuff() {
        return loadBalanced.getForObject("http://stores/stores", String.class);
    }

    public String doStuff() {
        return restTemplate.getForObject("http://example.com", String.class);
    }
}
```

Notice the use of the `@Primary` annotation on the plain `RestTemplate` declaration in the preceding example to disambiguate the unqualified `@Autowired` injection.

If you see errors such as `java.lang.IllegalArgumentException: Can not set org.springframework.web.client.RestTemplate field com.my.app.Foo.restTemplate to com.sun.proxy.$Proxy89`, try injecting `RestOperations` or setting `spring.aop.proxyTargetClass=true`.

38.6. Multiple WebClient Objects

If you want a `WebClient` that is not load-balanced, create a `WebClient` bean and inject it. To access the load-balanced `WebClient`, use the `@LoadBalanced` qualifier when you create your `@Bean`, as the following example shows:

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    WebClient.Builder loadBalanced() {
        return WebClient.builder();
    }

    @Primary
    @Bean
    WebClient.Builder webClient() {
        return WebClient.builder();
    }
}

public class MyClass {
    @Autowired
    private WebClient.Builder webClientBuilder;

    @Autowired
    @LoadBalanced
    private WebClient.Builder loadBalanced;

    public Mono<String> doOtherStuff() {
        return loadBalanced.build().get().uri("http://stores/stores")
                        .retrieve().bodyToMono(String.class);
    }

    public Mono<String> doStuff() {
        return webClientBuilder.build().get().uri("http://example.com")
                        .retrieve().bodyToMono(String.class);
    }
}
```

38.7. Spring WebFlux `WebClient` as a Load Balancer Client

The Spring WebFlux can work with both reactive and non-reactive `WebClient` configurations, as the topics describe:

- Spring WebFlux `WebClient` with `ReactorLoadBalancerExchangeFilterFunction`
- [load-balancer-exchange-filter-functionload-balancer-exchange-filter-function]

38.7.1. Spring WebFlux `WebClient` with `ReactorLoadBalancerExchangeFilterFunction`

You can configure `WebClient` to use the `ReactiveLoadBalancer`. If you add Spring Cloud LoadBalancer starter to your project and if `spring-webflux` is on the classpath, `ReactorLoadBalancerExchangeFilterFunction` is auto-configured. The following example shows how to configure a `WebClient` to use reactive load-balancer:

```
public class MyClass {
    @Autowired
    private ReactorLoadBalancerExchangeFilterFunction lbFunction;

    public Mono<String> doOtherStuff() {
        return WebClient.builder().baseUrl("http://stores")
            .filter(lbFunction)
            .build()
            .get()
            .uri("/stores")
            .retrieve()
            .bodyToMono(String.class);
    }
}
```

The URI needs to use a virtual host name (that is, a service name, not a host name). The `ReactorLoadBalancer` is used to create a full physical address.

By default, if you have `spring-cloud-netflix-ribbon` in your classpath, `LoadBalancerExchangeFilterFunction` is used to maintain backward compatibility. To use `ReactorLoadBalancerExchangeFilterFunction`, set the `spring.cloud.loadbalancer.ribbon.enabled` property to `false`.

38.7.2. Spring WebFlux `WebClient` with a Non-reactive Load Balancer Client

If you you do not have Spring Cloud LoadBalancer starter in your project but you do have spring-cloud-starter-netflix-ribbon, you can still use `WebClient` with `LoadBalancerClient`. If `spring-webflux` is on the classpath, `LoadBalancerExchangeFilterFunction` is auto-configured. Note, however, that this uses a non-reactive client under the hood. The following example shows how to configure a `WebClient` to use load-balancer:

```java
public class MyClass {
    @Autowired
    private LoadBalancerExchangeFilterFunction lbFunction;

    public Mono<String> doOtherStuff() {
        return WebClient.builder().baseUrl("http://stores")
            .filter(lbFunction)
            .build()
            .get()
            .uri("/stores")
            .retrieve()
            .bodyToMono(String.class);
    }
}
```

The URI needs to use a virtual host name (that is, a service name, not a host name). The `LoadBalancerClient` is used to create a full physical address.

WARN: This approach is now deprecated. We suggest that you use WebFlux with reactive Load-Balancer instead.

38.8. Ignore Network Interfaces

Sometimes, it is useful to ignore certain named network interfaces so that they can be excluded from Service Discovery registration (for example, when running in a Docker container). A list of regular expressions can be set to cause the desired network interfaces to be ignored. The following configuration ignores the `docker0` interface and all interfaces that start with `veth`:

Example 2. application.yml

```yaml
spring:
  cloud:
    inetutils:
      ignoredInterfaces:
        - docker0
        - veth.*
```

You can also force the use of only specified network addresses by using a list of regular expressions, as the following example shows:

Example 3. bootstrap.yml

```
spring:
  cloud:
    inetutils:
      preferredNetworks:
        - 192.168
        - 10.0
```

You can also force the use of only site-local addresses, as the following example shows:

Example 4. application.yml

```
spring:
  cloud:
    inetutils:
      useOnlySiteLocalInterfaces: true
```

See Inet4Address.html.isSiteLocalAddress() for more details about what constitutes a site-local address.

38.9. HTTP Client Factories

Spring Cloud Commons provides beans for creating both Apache HTTP clients (`ApacheHttpClientFactory`) and OK HTTP clients (`OkHttpClientFactory`). The `OkHttpClientFactory` bean is created only if the OK HTTP jar is on the classpath. In addition, Spring Cloud Commons provides beans for creating the connection managers used by both clients: `ApacheHttpClientConnectionManagerFactory` for the Apache HTTP client and `OkHttpClientConnectionPoolFactory` for the OK HTTP client. If you would like to customize how the HTTP clients are created in downstream projects, you can provide your own implementation of these beans. In addition, if you provide a bean of type `HttpClientBuilder` or `OkHttpClient.Builder`, the default factories use these builders as the basis for the builders returned to downstream projects. You can also disable the creation of these beans by setting `spring.cloud.httpclientfactories.apache.enabled` or `spring.cloud.httpclientfactories.ok.enabled` to `false`.

38.10. Enabled Features

Spring Cloud Commons provides a `/features` actuator endpoint. This endpoint returns features available on the classpath and whether they are enabled. The information returned includes the feature type, name, version, and vendor.

38.10.1. Feature types

There are two types of 'features': abstract and named.

Abstract features are features where an interface or abstract class is defined and that an implementation the creates, such as `DiscoveryClient`, `LoadBalancerClient`, or `LockService`. The abstract class or interface is used to find a bean of that type in the context. The version displayed is `bean.getClass().getPackage().getImplementationVersion()`.

Named features are features that do not have a particular class they implement. These features include “Circuit Breaker”, “API Gateway”, “Spring Cloud Bus”, and others. These features require a name and a bean type.

38.10.2. Declaring features

Any module can declare any number of `HasFeature` beans, as the following examples show:

```
@Bean
public HasFeatures commonsFeatures() {
  return HasFeatures.abstractFeatures(DiscoveryClient.class,
LoadBalancerClient.class);
}

@Bean
public HasFeatures consulFeatures() {
  return HasFeatures.namedFeatures(
    new NamedFeature("Spring Cloud Bus", ConsulBusAutoConfiguration.class),
    new NamedFeature("Circuit Breaker", HystrixCommandAspect.class));
}

@Bean
HasFeatures localFeatures() {
  return HasFeatures.builder()
      .abstractFeature(Something.class)
      .namedFeature(new NamedFeature("Some Other Feature", Someother.class))
      .abstractFeature(Somethingelse.class)
      .build();
}
```

Each of these beans should go in an appropriately guarded `@Configuration`.

38.11. Spring Cloud Compatibility Verification

Due to the fact that some users have problem with setting up Spring Cloud application, we’ve decided to add a compatibility verification mechanism. It will break if your current setup is not compatible with Spring Cloud requirements, together with a report, showing what exactly went wrong.

At the moment we verify which version of Spring Boot is added to your classpath.

Example of a report

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


Description:

Your project setup is incompatible with our requirements due to following reasons:

- Spring Boot [2.1.0.RELEASE] is not compatible with this Spring Cloud release
train


Action:

Consider applying the following actions:

- Change Spring Boot version to one of the following versions [1.2.x, 1.3.x] .
You can find the latest Spring Boot versions here
[https://spring.io/projects/spring-boot#learn].
If you want to learn more about the Spring Cloud Release train compatibility, you
can visit this page [https://spring.io/projects/spring-cloud#overview] and check
the [Release Trains] section.
```

In order to disable this feature, set `spring.cloud.compatibility-verifier.enabled` to `false`. If you want to override the compatible Spring Boot versions, just set the `spring.cloud.compatibility-verifier.compatible-boot-versions` property with a comma separated list of compatible Spring Boot versions.

Chapter 39. Spring Cloud LoadBalancer

Spring Cloud provides its own client-side load-balancer abstraction and implementation. For the load-balancing mechanism, `ReactiveLoadBalancer` interface has been added and a Round-Robin-based implementation has been provided for it. In order to get instances to select from reactive `ServiceInstanceListSupplier` is used. Currently we support a service-discovery-based implementation of `ServiceInstanceListSupplier` that retrieves available instances from Service Discovery using a Discovery Client available in the classpath.

39.1. Spring Cloud LoadBalancer integrations

In order to make it easy to use Spring Cloud LoadBalancer, we provide `ReactorLoadBalancerExchangeFilterFunction` that can be used with `WebClient` and `BlockingLoadBalancerClient` that works with `RestTemplate`. You can see more information and examples of usage in the following sections:

- Spring RestTemplate as a Load Balancer Client
- Spring WebClient as a Load Balancer Client
- Spring WebFlux WebClient with `ReactorLoadBalancerExchangeFilterFunction`

39.2. Spring Cloud LoadBalancer Caching

Apart from the basic `ServiceInstanceListSupplier` implementation that retrieves instances via `DiscoveryClient` each time it has to choose an instance, we provide two caching implementations.

39.2.1. Caffeine-backed LoadBalancer Cache Implementation

If you have `com.github.ben-manes.caffeine:caffeine` in the classpath, Caffeine-based implementation will be used. See the LoadBalancerCacheConfiguration section for information on how to configure it.

If you are using Caffeine, you can also override the default Caffeine Cache setup for the LoadBalancer by passing your own Caffeine Specification in the `spring.cloud.loadbalancer.cache.caffeine.spec` property.

WARN: Passing your own Caffeine specification will override any other LoadBalancerCache settings, including General LoadBalancer Cache Configuration fields, such as `ttl` and `capacity`.

39.2.2. Default LoadBalancer Cache Implementation

If you do not have Caffeine in the classpath, the `DefaultLoadBalancerCache`, which comes automatically with `spring-cloud-starter-loadbalancer`, will be used. See the LoadBalancerCacheConfiguration section for information on how to configure it.

To use Caffeine instead of the default cache, add the `com.github.ben-manes.caffeine:caffeine` dependency to classpath.

39.2.3. LoadBalancer Cache Configuration

You can set your own `ttl` value (the time after write after which entries should be expired), expressed as `Duration`, by passing a `String` compliant with the Spring Boot `String` to `Duration` converter syntax. as the value of the `spring.cloud.loadbalancer.cache.ttl` property. You can also set your own LoadBalancer cache initial capacity by setting the value of the `spring.cloud.loadbalancer.cache.capacity` property.

The default setup includes `ttl` set to 35 seconds and the default `initialCapacity` is `256`.

You can also altogether disable loadBalancer caching by setting the value of `spring.cloud.loadbalancer.cache.enabled` to `false`.

Although the basic, non-cached, implementation is useful for prototyping and testing, it's much less efficient than the cached versions, so we recommend always using the cached version in production.

39.3. Zone-Based Load-Balancing

To enable zone-based load-balancing, we provide the `ZonePreferenceServiceInstanceListSupplier`. We use `DiscoveryClient`-specific `zone` configuration (for example, `eureka.instance.metadata-map.zone`) to pick the zone that the client tries to filter available service instances for.

You can also override `DiscoveryClient`-specific zone setup by setting the value of `spring.cloud.loadbalancer.zone` property.

For the time being, only Eureka Discovery Client is instrumented to set the LoadBalancer zone. For other discovery client, set the `spring.cloud.loadbalancer.zone` property. More instrumentations coming shortly.

To determine the zone of a retrieved `ServiceInstance`, we check the value under the `"zone"` key in its metadata map.

The `ZonePreferenceServiceInstanceListSupplier` filters retrieved instances and only returns the ones within the same zone. If the zone is `null` or there are no instances within the same zone, it returns all the retrieved instances.

In order to use the zone-based load-balancing approach, you will have to instantiate a `ZonePreferenceServiceInstanceListSupplier` bean in a custom configuration.

We use delegates to work with `ServiceInstanceListSupplier` beans. We suggest passing a `DiscoveryClientServiceInstanceListSupplier` delegate in the constructor of `ZonePreferenceServiceInstanceListSupplier` and, in turn, wrapping the latter with a `CachingServiceInstanceListSupplier` to leverage LoadBalancer caching mechanism.

You could use this sample configuration to set it up:

```
public class CustomLoadBalancerConfiguration {

    @Bean
    public ServiceInstanceListSupplier discoveryClientServiceInstanceListSupplier(
            ConfigurableApplicationContext context) {
        return ServiceInstanceListSuppliers.builder()
                    .withDiscoveryClient()
                    .withZonePreference()
                    .withCaching()
                    .build(context);
    }
}
```

39.4. Instance Health-Check for LoadBalancer

It is possible to enable a scheduled HealthCheck for the LoadBalancer. The `HealthCheckServiceInstanceListSupplier` is provided for that. It regularly verifies if the instances provided by a delegate `ServiceInstanceListSupplier` are still alive and only returns the healthy instances, unless there are none - then it returns all the retrieved instances.

This mechanism is particularly helpful while using the `SimpleDiscoveryClient`. For the clients backed by an actual Service Registry, it's not necessary to use, as we already get healthy instances after querying the external ServiceDiscovery.

TIP

This supplier is also recommended for setups with a small number of instances per service in order to avoid retrying calls on a failing instance.

`HealthCheckServiceInstanceListSupplier` uses properties prefixed with `spring.cloud.loadbalancer.health-check`. You can set the `initialDelay` and `interval` for the scheduler. You can set the default path for the healthcheck URL by setting the value of the `spring.cloud.loadbalancer.health-check.path.default`. You can also set a specific value for any given service by setting the value of the `spring.cloud.loadbalancer.health-check.path.[SERVICE_ID]`, substituting the `[SERVICE_ID]` with the correct ID of your service. If the path is not set, `/actuator/health` is used by default.

TIP

If you rely on the default path (`/actuator/health`), make sure you add `spring-boot-starter-actuator` to your collaborator's dependencies, unless you are planning to add such an endpoint on your own.

In order to use the health-check scheduler approach, you will have to instantiate a `HealthCheckServiceInstanceListSupplier` bean in a custom configuration.

We use delegates to work with `ServiceInstanceListSupplier` beans. We suggest passing a `DiscoveryClientServiceInstanceListSupplier` delegate in the constructor of `HealthCheckServiceInstanceListSupplier`.

You could use this sample configuration to set it up:

```
public class CustomLoadBalancerConfiguration {

    @Bean
    public ServiceInstanceListSupplier discoveryClientServiceInstanceListSupplier(
            ConfigurableApplicationContext context) {
        return ServiceInstanceListSupplier.builder()
                    .withDiscoveryClient()
                    .withHealthChecks()
                    .build(context);
        }
    }
```

NOTE

`HealthCheckServiceInstanceListSupplier` has its own caching mechanism based on Reactor Flux `replay()`, therefore, if it's being used, you may want to skip wrapping that supplier with `CachingServiceInstanceListSupplier`.

39.5. Spring Cloud LoadBalancer Starter

We also provide a starter that allows you to easily add Spring Cloud LoadBalancer in a Spring Boot app. In order to use it, just add `org.springframework.cloud:spring-cloud-starter-loadbalancer` to your Spring Cloud dependencies in your build file.

Spring Cloud LoadBalancer starter includes Spring Boot Caching and Evictor.

If you have both Ribbon and Spring Cloud LoadBalancer int the classpath, in order to maintain backward compatibility, Ribbon-based implementations will be used by default. In order to switch to using Spring Cloud LoadBalancer under the hood, make sure you set the property `spring.cloud.loadbalancer.ribbon.enabled` to `false`.

39.6. Passing Your Own Spring Cloud LoadBalancer Configuration

You can also use the `@LoadBalancerClient` annotation to pass your own load-balancer client configuration, passing the name of the load-balancer client and the configuration class, as follows:

```
@Configuration
@LoadBalancerClient(value = "stores", configuration =
CustomLoadBalancerConfiguration.class)
public class MyConfiguration {

    @Bean
    @LoadBalanced
    public WebClient.Builder loadBalancedWebClientBuilder() {
        return WebClient.builder();
    }
}
```

TIP

In order to make working on your own LoadBalancer configuration easier, we have added a `builder()` method to the `ServiceInstanceListSupplier` class.

TIP

You can also use our alternative predefined configurations in place of the default ones by setting the value of `spring.cloud.loadbalancer.configurations` property to `zone-preference` to use `ZonePreferenceServiceInstanceListSupplier` with caching or to `health-check` to use `HealthCheckServiceInstanceListSupplier` with caching.

You can use this feature to instantiate different implementations of `ServiceInstanceListSupplier` or `ReactorLoadBalancer`, either written by you, or provided by us as alternatives (for example `ZonePreferenceServiceInstanceListSupplier`) to override the default setup.

You can see an example of a custom configuration here.

The annotation `value` arguments (`stores` in the example above) specifies the service id of the service that we should send the requests to with the given custom configuration.

You can also pass multiple configurations (for more than one load-balancer client) through the `@LoadBalancerClients` annotation, as the following example shows:

```
@Configuration
@LoadBalancerClients({@LoadBalancerClient(value = "stores", configuration =
StoresLoadBalancerClientConfiguration.class), @LoadBalancerClient(value =
"customers", configuration = CustomersLoadBalancerClientConfiguration.class)})
public class MyConfiguration {

    @Bean
    @LoadBalanced
    public WebClient.Builder loadBalancedWebClientBuilder() {
        return WebClient.builder();
    }
}
```

Chapter 40. Spring Cloud Circuit Breaker

40.1. Introduction

Spring Cloud Circuit breaker provides an abstraction across different circuit breaker implementations. It provides a consistent API to use in your applications, letting you, the developer, choose the circuit breaker implementation that best fits your needs for your application.

40.1.1. Supported Implementations

Spring Cloud supports the following circuit-breaker implementations:

- Netflix Hystrix
- Resilience4J
- Sentinel
- Spring Retry

40.2. Core Concepts

To create a circuit breaker in your code, you can use the `CircuitBreakerFactory` API. When you include a Spring Cloud Circuit Breaker starter on your classpath, a bean that implements this API is automatically created for you. The following example shows a simple example of how to use this API:

```
@Service
public static class DemoControllerService {
    private RestTemplate rest;
    private CircuitBreakerFactory cbFactory;

    public DemoControllerService(RestTemplate rest, CircuitBreakerFactory
cbFactory) {
        this.rest = rest;
        this.cbFactory = cbFactory;
    }

    public String slow() {
        return cbFactory.create("slow").run(() -> rest.getForObject("/slow",
String.class), throwable -> "fallback");
    }

}
```

The `CircuitBreakerFactory.create` API creates an instance of a class called `CircuitBreaker`. The `run` method takes a `Supplier` and a `Function`. The `Supplier` is the code that you are going to wrap in a

circuit breaker. The `Function` is the fallback that is executed if the circuit breaker is tripped. The function is passed the `Throwable` that caused the fallback to be triggered. You can optionally exclude the fallback if you do not want to provide one.

40.2.1. Circuit Breakers In Reactive Code

If Project Reactor is on the class path, you can also use `ReactiveCircuitBreakerFactory` for your reactive code. The following example shows how to do so:

```
@Service
public static class DemoControllerService {
    private ReactiveCircuitBreakerFactory cbFactory;
    private WebClient webClient;


    public DemoControllerService(WebClient webClient,
ReactiveCircuitBreakerFactory cbFactory) {
        this.webClient = webClient;
        this.cbFactory = cbFactory;
    }

    public Mono<String> slow() {
        return
webClient.get().uri("/slow").retrieve().bodyToMono(String.class).transform(
        it -> cbFactory.create("slow").run(it, throwable -> return
Mono.just("fallback")));
    }
}
```

The `ReactiveCircuitBreakerFactory.create` API creates an instance of a class called `ReactiveCircuitBreaker`. The `run` method takes a `Mono` or a `Flux` and wraps it in a circuit breaker. You can optionally profile a fallback `Function`, which will be called if the circuit breaker is tripped and is passed the `Throwable` that caused the failure.

40.3. Configuration

You can configure your circuit breakers by creating beans of type `Customizer`. The `Customizer` interface has a single method (called `customize`) that takes the `Object` to customize.

For detailed information on how to customize a given implementation see the following documentation:

- Hystrix
- Resilience4J
- Sentinal
- Spring Retry

Some `CircuitBreaker` implementations such as `Resilience4JCircuitBreaker` call `customize` method every time `CircuitBreaker#run` is called. It can be inefficient. In that case, you can use `CircuitBreaker#once` method. It is useful where calling `customize` many times doesn't make sense, for example, in case of consuming Resilience4j's events.

The following example shows the way for each `io.github.resilience4j.circuitbreaker.CircuitBreaker` to consume events.

```
Customizer.once(circuitBreaker -> {
  circuitBreaker.getEventPublisher()
    .onStateTransition(event -> log.info("{}: {}", event.getCircuitBreakerName(),
event.getStateTransition()));
}, CircuitBreaker::getName)
```

Chapter 41. CachedRandomPropertySource

Spring Cloud Context provides a `PropertySource` that caches random values based on a key. Outside of the caching functionality it works the same as Spring Boot's `RandomValuePropertySource`. This random value might be useful in the case where you want a random value that is consistent even after the Spring Application context restarts. The property value takes the form of `cachedrandom.[yourkey].[type]` where `yourkey` is the key in the cache. The `type` value can be any type supported by Spring Boot's `RandomValuePropertySource`.

```
myrandom=${cachedrandom.appname.value}
```

Chapter 42. Configuration Properties

To see the list of all Spring Cloud Commons related configuration properties please check the Appendix page.

Spring Cloud Config

Hoxton.SR8

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system. With the Config Server, you have a central place to manage external properties for applications across all environments. The concepts on both client and server map identically to the Spring `Environment` and `PropertySource` abstractions, so they fit very well with Spring applications but can be used with any application running in any language. As an application moves through the deployment pipeline from dev to test and into production, you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate. The default implementation of the server storage backend uses git, so it easily supports labelled versions of configuration environments as well as being accessible to a wide range of tooling for managing the content. It is easy to add alternative implementations and plug them in with Spring configuration.

Chapter 43. Quick Start

This quick start walks through using both the server and the client of Spring Cloud Config Server.

First, start the server, as follows:

```
$ cd spring-cloud-config-server
$ ../mvnw spring-boot:run
```

The server is a Spring Boot application, so you can run it from your IDE if you prefer to do so (the main class is `ConfigServerApplication`).

Next try out a client, as follows:

```
$ curl localhost:8888/foo/development
{"name":"foo","label":"master","propertySources":[
  {"name":"https://github.com/scratches/config-repo/foo-
development.properties","source":{"bar":"spam"}},
  {"name":"https://github.com/scratches/config-
repo/foo.properties","source":{"foo":"bar"}}
]}
```

The default strategy for locating property sources is to clone a git repository (at `spring.cloud.config.server.git.uri`) and use it to initialize a mini `SpringApplication`. The mini-application's `Environment` is used to enumerate property sources and publish them at a JSON endpoint.

The HTTP service has resources in the following form:

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

where `application` is injected as the `spring.config.name` in the `SpringApplication` (what is normally `application` in a regular Spring Boot app), `profile` is an active profile (or comma-separated list of properties), and `label` is an optional git label (defaults to `master`.)

Spring Cloud Config Server pulls configuration for remote clients from various sources. The following example gets configuration from a git repository (which must be provided), as shown in the following example:

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
```

Other sources are any JDBC compatible database, Subversion, Hashicorp Vault, Credhub and local filesystems.

43.1. Client Side Usage

To use these features in an application, you can build it as a Spring Boot application that depends on spring-cloud-config-client (for an example, see the test cases for the config-client or the sample application). The most convenient way to add the dependency is with a Spring Boot starter `org.springframework.cloud:spring-cloud-starter-config`. There is also a parent pom and BOM (`spring-cloud-starter-parent`) for Maven users and a Spring IO version management properties file for Gradle and Spring CLI users. The following example shows a typical Maven configuration:

pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>{spring-boot-docs-version}</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>{spring-cloud-version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

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

<!-- repositories also needed for snapshots and milestones -->
```

Now you can create a standard Spring Boot application, such as the following HTTP server:

```
@SpringBootApplication
@RestController
public class Application {

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

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

}
```

When this HTTP server runs, it picks up the external configuration from the default local config server (if it is running) on port 8888. To modify the startup behavior, you can change the location of the config server by using `bootstrap.properties` (similar to `application.properties` but for the bootstrap phase of an application context), as shown in the following example:

```
spring.cloud.config.uri: http://myconfigserver.com
```

By default, if no application name is set, `application` will be used. To modify the name, the following property can be added to the `bootstrap.properties` file:

```
spring.application.name: myapp
```

When setting the property `${spring.application.name}` do not prefix your app name with the reserved word `application-` to prevent issues resolving the correct property source.

The bootstrap properties show up in the `/env` endpoint as a high-priority property source, as shown in the following example.

```
$ curl localhost:8080/env
{
  "profiles":[],
  "configService:https://github.com/spring-cloud-samples/config-
repo/bar.properties":{"foo":"bar"},
  "servletContextInitParams":{},
  "systemProperties":{...},
  ...
}
```

A property source called `configService:<URL of remote repository>/<file name>` contains the `foo`

property with a value of `bar` and is the highest priority.

The URL in the property source name is the git repository, not the config server URL.

Chapter 44. Spring Cloud Config Server

Spring Cloud Config Server provides an HTTP resource-based API for external configuration (name-value pairs or equivalent YAML content). The server is embeddable in a Spring Boot application, by using the `@EnableConfigServer` annotation. Consequently, the following application is a config server:

ConfigServer.java

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

Like all Spring Boot applications, it runs on port 8080 by default, but you can switch it to the more conventional port 8888 in various ways. The easiest, which also sets a default configuration repository, is by launching it with `spring.config.name=configserver` (there is a `configserver.yml` in the Config Server jar). Another is to use your own `application.properties`, as shown in the following example:

application.properties

```
server.port: 8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

where `${user.home}/config-repo` is a git repository containing YAML and properties files.

On Windows, you need an extra "/" in the file URL if it is absolute with a drive prefix (for example,`/${user.home}/config-repo`).

The following listing shows a recipe for creating the git repository in the preceding example:

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo info.foo: bar > application.properties
$ git add -A .
$ git commit -m "Add application.properties"
```

Using the local filesystem for your git repository is intended for testing only. You should use a server to host your configuration repositories in production.

The initial clone of your configuration repository can be quick and efficient if you keep only text files in it. If you store binary files, especially large ones, you may experience delays on the first request for configuration or encounter out of memory errors in the server.

44.1. Environment Repository

Where should you store the configuration data for the Config Server? The strategy that governs this behaviour is the `EnvironmentRepository`, serving `Environment` objects. This `Environment` is a shallow copy of the domain from the Spring `Environment` (including `propertySources` as the main feature). The `Environment` resources are parametrized by three variables:

- `{application}`, which maps to `spring.application.name` on the client side.
- `{profile}`, which maps to `spring.profiles.active` on the client (comma-separated list).
- `{label}`, which is a server side feature labelling a "versioned" set of config files.

Repository implementations generally behave like a Spring Boot application, loading configuration files from a `spring.config.name` equal to the `{application}` parameter, and `spring.profiles.active` equal to the `{profiles}` parameter. Precedence rules for profiles are also the same as in a regular Spring Boot application: Active profiles take precedence over defaults, and, if there are multiple profiles, the last one wins (similar to adding entries to a `Map`).

The following sample client application has this bootstrap configuration:

bootstrap.yml

```
spring:
  application:
    name: foo
  profiles:
    active: dev,mysql
```

(As usual with a Spring Boot application, these properties could also be set by environment variables or command line arguments).

If the repository is file-based, the server creates an `Environment` from `application.yml` (shared between all clients) and `foo.yml` (with `foo.yml` taking precedence). If the YAML files have documents inside them that point to Spring profiles, those are applied with higher precedence (in order of the profiles listed). If there are profile-specific YAML (or properties) files, these are also applied with higher precedence than the defaults. Higher precedence translates to a `PropertySource` listed earlier in the `Environment`. (These same rules apply in a standalone Spring Boot application.)

You can set spring.cloud.config.server.accept-empty to false so that Server would return a HTTP 404 status, if the application is not found.By default, this flag is set to true.

44.1.1. Git Backend

The default implementation of `EnvironmentRepository` uses a Git backend, which is very convenient for managing upgrades and physical environments and for auditing changes. To change the location of the repository, you can set the `spring.cloud.config.server.git.uri` configuration property in the Config Server (for example in `application.yml`). If you set it with a `file:` prefix, it should work from a local repository so that you can get started quickly and easily without a server. However, in that case, the server operates directly on the local repository without cloning it (it does not matter if it is not bare because the Config Server never makes changes to the "remote" repository). To scale the Config Server up and make it highly available, you need to have all instances of the server pointing to the same repository, so only a shared file system would work. Even in that case, it is better to use the `ssh:` protocol for a shared filesystem repository, so that the server can clone it and use a local working copy as a cache.

This repository implementation maps the `{label}` parameter of the HTTP resource to a git label (commit id, branch name, or tag). If the git branch or tag name contains a slash (`/`), then the label in the HTTP URL should instead be specified with the special string `(_)` (to avoid ambiguity with other URL paths). For example, if the label is `foo/bar`, replacing the slash would result in the following label: `foo(_)bar`. The inclusion of the special string `(_)` can also be applied to the `{application}` parameter. If you use a command-line client such as curl, be careful with the brackets in the URL — you should escape them from the shell with single quotes ('').

Skipping SSL Certificate Validation

The configuration server's validation of the Git server's SSL certificate can be disabled by setting the `git.skipSslValidation` property to `true` (default is `false`).

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://example.com/my/repo
          skipSslValidation: true
```

Setting HTTP Connection Timeout

You can configure the time, in seconds, that the configuration server will wait to acquire an HTTP connection. Use the `git.timeout` property.

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://example.com/my/repo
          timeout: 4
```

Placeholders in Git URI

Spring Cloud Config Server supports a git repository URL with placeholders for the `{application}` and `{profile}` (and `{label}` if you need it, but remember that the label is applied as a git label anyway). So you can support a "one repository per application" policy by using a structure similar to the following:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/myorg/{application}
```

You can also support a "one repository per profile" policy by using a similar pattern but with `{profile}`.

Additionally, using the special string "(_)" within your `{application}` parameters can enable support for multiple organizations, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/{application}
```

where `{application}` is provided at request time in the following format: `organization(_)application`.

Pattern Matching and Multiple Repositories

Spring Cloud Config also includes support for more complex requirements with pattern matching on the application and profile name. The pattern format is a comma-separated list of `{application}/{profile}` names with wildcards (note that a pattern beginning with a wildcard may need to be quoted), as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            simple: https://github.com/simple/config-repo
            special:
              pattern: special*/dev*,*special*/dev*
              uri: https://github.com/special/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

If `{application}/{profile}` does not match any of the patterns, it uses the default URI defined under `spring.cloud.config.server.git.uri`. In the above example, for the "simple" repository, the pattern is `simple/*` (it only matches one application named `simple` in all profiles). The "local" repository matches all application names beginning with `local` in all profiles (the `/*` suffix is added automatically to any pattern that does not have a profile matcher).

The "one-liner" short cut used in the "simple" example can be used only if the only property to be set is the URI. If you need to set anything else (credentials, pattern, and so on) you need to use the full form.

The `pattern` property in the repo is actually an array, so you can use a YAML array (or `[0]`, `[1]`, etc. suffixes in properties files) to bind to multiple patterns. You may need to do so if you are going to run apps with multiple profiles, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            development:
              pattern:
                - '*/development'
                - '*/staging'
              uri: https://github.com/development/config-repo
            staging:
              pattern:
                - '*/qa'
                - '*/production'
              uri: https://github.com/staging/config-repo
```

Spring Cloud guesses that a pattern containing a profile that does not end in `*` implies that you actually want to match a list of profiles starting with this pattern (so `*/staging` is a shortcut for `["*/staging", "*/staging,*"]`, and so on). This is common where, for instance, you need to run applications in the “development” profile locally but also the “cloud” profile remotely.

Every repository can also optionally store config files in sub-directories, and patterns to search for those directories can be specified as `searchPaths`. The following example shows a config file at the top level:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: foo,bar*
```

In the preceding example, the server searches for config files in the top level and in the `foo/` sub-directory and also any sub-directory whose name begins with `bar`.

By default, the server clones remote repositories when configuration is first requested. The server can be configured to clone the repositories at startup, as shown in the following top-level example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          repos:
            team-a:
                pattern: team-a-*
                cloneOnStart: true
                uri: https://git/team-a/config-repo.git
            team-b:
                pattern: team-b-*
                cloneOnStart: false
                uri: https://git/team-b/config-repo.git
            team-c:
                pattern: team-c-*
                uri: https://git/team-a/config-repo.git
```

In the preceding example, the server clones team-a’s config-repo on startup, before it accepts any requests. All other repositories are not cloned until configuration from the repository is requested.

Setting a repository to be cloned when the Config Server starts up can help to identify a misconfigured configuration source (such as an invalid repository URI) quickly, while the Config Server is starting up. With `cloneOnStart` not enabled for a configuration source, the Config Server may start successfully with a misconfigured or invalid configuration source and not detect an error until an application requests configuration from that configuration source.

Authentication

To use HTTP basic authentication on the remote repository, add the `username` and `password` properties separately (not in the URL), as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          username: trolley
          password: strongpassword
```

If you do not use HTTPS and user credentials, SSH should also work out of the box when you store keys in the default directories (`~/.ssh`) and the URI points to an SSH location, such as `git@github.com:configuration/cloud-configuration`. It is important that an entry for the Git server be present in the `~/.ssh/known_hosts` file and that it is in `ssh-rsa` format. Other formats (such as `ecdsa-sha2-nistp256`) are not supported. To avoid surprises, you should ensure that only one entry is present in the `known_hosts` file for the Git server and that it matches the URL you provided to the config server. If you use a hostname in the URL, you want to have exactly that (not the IP) in the `known_hosts` file. The repository is accessed by using JGit, so any documentation you find on that should be applicable. HTTPS proxy settings can be set in `~/.git/config` or (in the same way as for any other JVM process) with system properties (`-Dhttps.proxyHost` and `-Dhttps.proxyPort`).

If you do not know where your `~/.git` directory is, use `git config --global` to manipulate the settings (for example, `git config --global http.sslVerify false`).

JGit requires RSA keys in PEM format. Below is an example ssh-keygen (from openssh) command that will generate a key in the corect format:

```
ssh-keygen -m PEM -t rsa -b 4096 -f ~/config_server_deploy_key.rsa
```

Warning: When working with SSH keys, the expected ssh private-key must begin with `-----BEGIN RSA PRIVATE KEY-----`. If the key starts with `-----BEGIN OPENSSH PRIVATE KEY-----` then the RSA key will not load when spring-cloud-config server is started. The error looks like:

```
- Error in object 'spring.cloud.config.server.git': codes
[PrivateKeyIsValid.spring.cloud.config.server.git,PrivateKeyIsValid]; arguments
[org.springframework.context.support.DefaultMessageSourceResolvable: codes
[spring.cloud.config.server.git.,]; arguments []; default message []]; default message
[Property 'spring.cloud.config.server.git.privateKey' is not a valid private key]
```

To correct the above error the RSA key must be converted to PEM format. An example using openssh is provided above for generating a new key in the appropriate format.

Authentication with AWS CodeCommit

Spring Cloud Config Server also supports AWS CodeCommit authentication. AWS CodeCommit uses an authentication helper when using Git from the command line. This helper is not used with the JGit library, so a JGit CredentialProvider for AWS CodeCommit is created if the Git URI matches the AWS CodeCommit pattern. AWS CodeCommit URIs follow this pattern://git-codecommit.${AWS_REGION}.amazonaws.com/${repopath}.

If you provide a username and password with an AWS CodeCommit URI, they must be the AWS accessKeyId and secretAccessKey that provide access to the repository. If you do not specify a username and password, the accessKeyId and secretAccessKey are retrieved by using the AWS Default Credential Provider Chain.

If your Git URI matches the CodeCommit URI pattern (shown earlier), you must provide valid AWS credentials in the username and password or in one of the locations supported by the default credential provider chain. AWS EC2 instances may use IAM Roles for EC2 Instances.

The `aws-java-sdk-core` jar is an optional dependency. If the `aws-java-sdk-core` jar is not on your classpath, the AWS Code Commit credential provider is not created, regardless of the git server URI.

Authentication with Google Cloud Source

Spring Cloud Config Server also supports authenticating against Google Cloud Source repositories.

If your Git URI uses the `http` or `https` protocol and the domain name is `source.developers.google.com`, the Google Cloud Source credentials provider will be used. A Google Cloud Source repository URI has the format `source.developers.google.com/p/${GCP_PROJECT}/r/${REPO}`. To obtain the URI for your repository, click on "Clone" in the Google Cloud Source UI, and select "Manually generated credentials". Do not generate any credentials, simply copy the displayed URI.

The Google Cloud Source credentials provider will use Google Cloud Platform application default credentials. See Google Cloud SDK documentation on how to create application default credentials for a system. This approach will work for user accounts in dev environments and for service accounts in production environments.

`com.google.auth:google-auth-library-oauth2-http` is an optional dependency. If the `google-auth-library-oauth2-http` jar is not on your classpath, the Google Cloud Source credential provider is not created, regardless of the git server URI.

Git SSH configuration using properties

By default, the JGit library used by Spring Cloud Config Server uses SSH configuration files such as `~/.ssh/known_hosts` and `/etc/ssh/ssh_config` when connecting to Git repositories by using an SSH URI. In cloud environments such as Cloud Foundry, the local filesystem may be ephemeral or not easily accessible. For those cases, SSH configuration can be set by using Java properties. In order to activate property-based SSH configuration, the `spring.cloud.config.server.git.ignoreLocalSshSettings` property must be set to `true`, as shown in the following example:

```
  spring:
    cloud:
      config:
        server:
          git:
            uri: git@gitserver.com:team/repo1.git
            ignoreLocalSshSettings: true
            hostKey: someHostKey
            hostKeyAlgorithm: ssh-rsa
            privateKey: |
                          -----BEGIN RSA PRIVATE KEY-----

MIIEpgIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqHp5AKaRbtAC3DqX

IXFMPgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCoqF

ol8+ngLqRZCyBtQN7zYByWMRirPGoDUqdPYrj2yq+ObBBNhg5N+hOwKjjpzdj2Ud

1l7R+wxIqmJo1IYyy16xS8WsjyQuyC0lL456qkd5BDZ0Ag8j2X9H9D5220Ln7s9i

oezTipXipS7p7Jekf3Ywx6abJwOmB0rX79dV4qiNcGgzATnG1PkXxqt76VhcGa0W

DDVHEEYGbSQ6hIGSh0I7BQun0aLRZojfE3gqHQIDAQABAoIBAQCZmGrk8BK6tXCd

fY6yTiKxFzwb38IQP0ojIUWNrq0+9Xt+NsypviLHkXfXXCKKU4zUHeIGVRq5MN9b

BO56/RrcQHHOoJdUWuOV2qMqJvPUtC0CpGkD+valhfD75MxoXU7s3FK7yjxy3rsG

EmfA6tHV8/4a5umo5TqSd2YTm5B19AhRqiuUVI1wTB41DjULUGiMYrnYrhzQlVvj

5MjnKTlYu3V8PoYDfv1GmxPPh6vlpafXEeEYN8VB97e5x3DGHjZ5UrurAmTLTdO8

+AahyoKsIY612TkkQthJlt7FJAwnCGMgY6podzzvzICLFmmTXYiZ/28I4BX/mOSe

pZVnfRixAoGBAO6Uiwt40/PKs53mCEWngslSCsh9oGAaLTf/XdvMns5VmuyyAyKG
```

```
ti8Ol5wqBMi4GIUzjbgUvSUt+IowIrG3f5tN85wpjQ1UGVcpTnl5Qo9xaS1PFScQ

xrtWZ9eNj2TsIAMp/svJsyGG3OibxfnuAIpSXNQiJPwRlW3irzpGgVx/AoGBANYW

dnhshUcEHMJi3aXwR12OTDnaLoanVGLwLnkqLSYUZA7ZegpKq90UAuBdcEfgdpyi

PhKpeaeIiAaNnFo8m9aoTKr+7I6/uMTlwrVnfrsVTZv3orxjwQV20YIBCVRKD1uX

VhE0ozPZxwwKSPAFocpyWpGHGreGF1AIYBE9UBtjAoGBAI8bfPgJpyFyMiGBjO6z

FwlJc/xlFqDusrcHL7abW5qq0L4v3R+FrJw3ZYufzLTVcKfdj6GelwJJO+8wBm+R

gTKYJItEhT48duLIfTDyIpHGVm9+I1MGhh5zKuCqIhxIYr9jHloBB7kRm0rPvYY4

VAykcNgyDvtAVODP+4m6JvhjAoGBALbtTqErKN47V0+JJpapLnF0KxGrqeGIjIRV

cYA6V4WYGr7NeIfesecfOC356PyhgPfpcVyEztwlvwTKb3RzIT1TZN8fH4YBr6Ee

KTbTjefRFhVUjQqnucAvfGi29f+9oE3Ei9f7wA+H35ocF6JvTYUsHNMIO/3gZ38N

CPjyCMa9AoGBAMhsITNe3QcbsXAbdUR00dDsIFVROzyFJ2m40i4KCRM35bC/BIBs

q0TY3we+ERB40U8Z2BvU61QuwaunJ2+uGadHo58VSVdggqAo0BSkH58innKKt96J
                         69pcVH/4rmLbXdcmNYGm6iu+MlPQk4BUZknHSmVHIFdJ0EPupVaQ8RHT
                         -----END RSA PRIVATE KEY-----
```


The following table describes the SSH configuration properties.

Table 2. SSH Configuration Properties

Property Name	Remarks
ignoreLocalSshSettings	If `true`, use property-based instead of file-based SSH config. Must be set at as `spring.cloud.config.server.git.ignoreLocalSshSettings`, **not** inside a repository definition.
privateKey	Valid SSH private key. Must be set if `ignoreLocalSshSettings` is true and Git URI is SSH format.
hostKey	Valid SSH host key. Must be set if `hostKeyAlgorithm` is also set.
hostKeyAlgorithm	One of `ssh-dss, ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384, or ecdsa-sha2-nistp521`. Must be set if `hostKey` is also set.
strictHostKeyChecking	`true` or `false`. If false, ignore errors with host key.
knownHostsFile	Location of custom `.known_hosts` file.

Property Name	Remarks
preferredAuthentications	Override server authentication method order. This should allow for evading login prompts if server has keyboard-interactive authentication before the `publickey` method.

Placeholders in Git Search Paths

Spring Cloud Config Server also supports a search path with placeholders for the `{application}` and `{profile}` (and `{label}` if you need it), as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: '{application}'
```

The preceding listing causes a search of the repository for files in the same name as the directory (as well as the top level). Wildcards are also valid in a search path with placeholders (any matching directory is included in the search).

Force pull in Git Repositories

As mentioned earlier, Spring Cloud Config Server makes a clone of the remote git repository in case the local copy gets dirty (for example, folder content changes by an OS process) such that Spring Cloud Config Server cannot update the local copy from remote repository.

To solve this issue, there is a `force-pull` property that makes Spring Cloud Config Server force pull from the remote repository if the local copy is dirty, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          force-pull: true
```

If you have a multiple-repositories configuration, you can configure the `force-pull` property per repository, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          force-pull: true
          repos:
            team-a:
                pattern: team-a-*
                uri: https://git/team-a/config-repo.git
                force-pull: true
            team-b:
                pattern: team-b-*
                uri: https://git/team-b/config-repo.git
                force-pull: true
            team-c:
                pattern: team-c-*
                uri: https://git/team-a/config-repo.git
```

The default value for `force-pull` property is `false`.

Deleting untracked branches in Git Repositories

As Spring Cloud Config Server has a clone of the remote git repository after check-outing branch to local repo (e.g fetching properties by label) it will keep this branch forever or till the next server restart (which creates new local repo). So there could be a case when remote branch is deleted but local copy of it is still available for fetching. And if Spring Cloud Config Server client service starts with `--spring.cloud.config.label=deletedRemoteBranch,master` it will fetch properties from `deletedRemoteBranch` local branch, but not from `master`.

In order to keep local repository branches clean and up to remote - `deleteUntrackedBranches` property could be set. It will make Spring Cloud Config Server **force** delete untracked branches from local repository. Example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          deleteUntrackedBranches: true
```

The default value for `deleteUntrackedBranches` property is `false`.

Git Refresh Rate

You can control how often the config server will fetch updated configuration data from your Git backend by using `spring.cloud.config.server.git.refreshRate`. The value of this property is specified in seconds. By default the value is 0, meaning the config server will fetch updated configuration from the Git repo every time it is requested.

44.1.2. Version Control Backend Filesystem Use

With VCS-based backends (git, svn), files are checked out or cloned to the local filesystem. By default, they are put in the system temporary directory with a prefix of `config-repo-`. On linux, for example, it could be `/tmp/config-repo-<randomid>`. Some operating systems routinely clean out temporary directories. This can lead to unexpected behavior, such as missing properties. To avoid this problem, change the directory that Config Server uses by setting `spring.cloud.config.server.git.basedir` or `spring.cloud.config.server.svn.basedir` to a directory that does not reside in the system temp structure.

44.1.3. File System Backend

There is also a “native” profile in the Config Server that does not use Git but loads the config files from the local classpath or file system (any static URL you want to point to with `spring.cloud.config.server.native.searchLocations`). To use the native profile, launch the Config Server with `spring.profiles.active=native`.

Remember to use the `file:` prefix for file resources (the default without a prefix is usually the classpath). As with any Spring Boot configuration, you can embed `${}` -style environment placeholders, but remember that absolute paths in Windows require an extra `/` (for example, `/${user.home}/config-repo`).

The default value of the `searchLocations` is identical to a local Spring Boot application (that is, `[classpath:/, classpath:/config, file:./, file:./config]`). This does not expose the `application.properties` from the server to all clients, because any property sources present in the server are removed before being sent to the client.

A filesystem backend is great for getting started quickly and for testing. To use it in production, you need to be sure that the file system is reliable and shared across all instances of the Config Server.

The search locations can contain placeholders for `{application}`, `{profile}`, and `{label}`. In this way, you can segregate the directories in the path and choose a strategy that makes sense for you (such as subdirectory per application or subdirectory per profile).

If you do not use placeholders in the search locations, this repository also appends the `{label}` parameter of the HTTP resource to a suffix on the search path, so properties files are loaded from

each search location **and** a subdirectory with the same name as the label (the labelled properties take precedence in the Spring Environment). Thus, the default behaviour with no placeholders is the same as adding a search location ending with `/{label}/`. For example, `file:/tmp/config` is the same as `file:/tmp/config,file:/tmp/config/{label}`. This behavior can be disabled by setting `spring.cloud.config.server.native.addLabelLocations=false`.

44.1.4. Vault Backend

Spring Cloud Config Server also supports Vault as a backend.

> Vault is a tool for securely accessing secrets. A secret is anything that to which you want to tightly control access, such as API keys, passwords, certificates, and other sensitive information. Vault provides a unified interface to any secret while providing tight access control and recording a detailed audit log.

For more information on Vault, see the Vault quick start guide.

To enable the config server to use a Vault backend, you can run your config server with the `vault` profile. For example, in your config server's `application.properties`, you can add `spring.profiles.active=vault`.

By default, the config server assumes that your Vault server runs at `127.0.0.1:8200`. It also assumes that the name of backend is `secret` and the key is `application`. All of these defaults can be configured in your config server's `application.properties`. The following table describes configurable Vault properties:

Name	Default Value
host	127.0.0.1
port	8200
scheme	http
backend	secret
defaultKey	application
profileSeparator	,
kvVersion	1
skipSslValidation	false
timeout	5
namespace	null

All of the properties in the preceding table must be prefixed with `spring.cloud.config.server.vault` or placed in the correct Vault section of a composite configuration.

All configurable properties can be found in

`org.springframework.cloud.config.server.environment.VaultEnvironmentProperties`.

Vault 0.10.0 introduced a versioned key-value backend (k/v backend version 2) that exposes a different API than earlier versions, it now requires a `data/` between the mount path and the actual context path and wraps secrets in a `data` object. Setting `spring.cloud.config.server.vault.kv-version=2` will take this into account.

Optionally, there is support for the Vault Enterprise `X-Vault-Namespace` header. To have it sent to Vault set the `namespace` property.

With your config server running, you can make HTTP requests to the server to retrieve values from the Vault backend. To do so, you need a token for your Vault server.

First, place some data in you Vault, as shown in the following example:

```
$ vault kv put secret/application foo=bar baz=bam
$ vault kv put secret/myapp foo=myappsbar
```

Second, make an HTTP request to your config server to retrieve the values, as shown in the following example:

`$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"`

You should see a response similar to the following:

```
{
   "name":"myapp",
   "profiles":[
      "default"
   ],
   "label":null,
   "version":null,
   "state":null,
   "propertySources":[
      {
         "name":"vault:myapp",
         "source":{
            "foo":"myappsbar"
         }
      },
      {
         "name":"vault:application",
         "source":{
            "baz":"bam",
            "foo":"bar"
         }
      }
   ]
}
```

The default way for a client to provide the necessary authentication to let Config Server talk to Vault is to set the X-Config-Token header. However, you can instead omit the header and configure the authentication in the server, by setting the same configuration properties as Spring Cloud Vault. The property to set is `spring.cloud.config.server.vault.authentication`. It should be set to one of the supported authentication methods. You may also need to set other properties specific to the authentication method you use, by using the same property names as documented for `spring.cloud.vault` but instead using the `spring.cloud.config.server.vault` prefix. See the Spring Cloud Vault Reference Guide for more detail.

If you omit the X-Config-Token header and use a server property to set the authentication, the Config Server application needs an additional dependency on Spring Vault to enable the additional authentication options. See the Spring Vault Reference Guide for how to add that dependency.

Multiple Properties Sources

When using Vault, you can provide your applications with multiple properties sources. For example, assume you have written data to the following paths in Vault:

```
secret/myApp,dev
secret/myApp
secret/application,dev
secret/application
```

Properties written to `secret/application` are available to all applications using the Config Server. An application with the name, `myApp`, would have any properties written to `secret/myApp` and `secret/application` available to it. When `myApp` has the `dev` profile enabled, properties written to all of the above paths would be available to it, with properties in the first path in the list taking priority over the others.

44.1.5. Accessing Backends Through a Proxy

The configuration server can access a Git or Vault backend through an HTTP or HTTPS proxy. This behavior is controlled for either Git or Vault by settings under `proxy.http` and `proxy.https`. These settings are per repository, so if you are using a composite environment repository you must configure proxy settings for each backend in the composite individually. If using a network which requires separate proxy servers for HTTP and HTTPS URLs, you can configure both the HTTP and the HTTPS proxy settings for a single backend.

The following table describes the proxy configuration properties for both HTTP and HTTPS proxies. All of these properties must be prefixed by `proxy.http` or `proxy.https`.

Table 3. Proxy Configuration Properties

Property Name	Remarks
host	The host of the proxy.
port	The port with which to access the proxy.
nonProxyHosts	Any hosts which the configuration server should access outside the proxy. If values are provided for both `proxy.http.nonProxyHosts` and `proxy.https.nonProxyHosts`, the `proxy.http` value will be used.
username	The username with which to authenticate to the proxy. If values are provided for both `proxy.http.username` and `proxy.https.username`, the `proxy.http` value will be used.
password	The password with which to authenticate to the proxy. If values are provided for both `proxy.http.password` and `proxy.https.password`, the `proxy.http` value will be used.

The following configuration uses an HTTPS proxy to access a Git repository.

```
spring:
  profiles:
    active: git
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          proxy:
            https:
              host: my-proxy.host.io
              password: myproxypassword
              port: '3128'
              username: myproxyusername
              nonProxyHosts: example.com
```

44.1.6. Sharing Configuration With All Applications

Sharing configuration between all applications varies according to which approach you take, as described in the following topics:

- File Based Repositories
- Vault Server

File Based Repositories

With file-based (git, svn, and native) repositories, resources with file names in `application*` (`application.properties`, `application.yml`, `application-*.properties`, and so on) are shared between all client applications. You can use resources with these file names to configure global defaults and have them be overridden by application-specific files as necessary.

The property overrides feature can also be used for setting global defaults, with placeholders applications allowed to override them locally.

With the “native” profile (a local file system backend) , you should use an explicit search location that is not part of the server’s own configuration. Otherwise, the `application*` resources in the default search locations get removed because they are part of the server.

Vault Server

When using Vault as a backend, you can share configuration with all applications by placing configuration in `secret/application`. For example, if you run the following Vault command, all applications using the config server will have the properties `foo` and `baz` available to them:

```
$ vault write secret/application foo=bar baz=bam
```

CredHub Server

When using CredHub as a backend, you can share configuration with all applications by placing configuration in `/application/` or by placing it in the `default` profile for the application. For example, if you run the following CredHub command, all applications using the config server will have the properties `shared.color1` and `shared.color2` available to them:

```
credhub set --name "/application/profile/master/shared" --type=json
value: {"shared.color1": "blue", "shared.color": "red"}
```

```
credhub set --name "/my-app/default/master/more-shared" --type=json
value: {"shared.word1": "hello", "shared.word2": "world"}
```

44.1.7. JDBC Backend

Spring Cloud Config Server supports JDBC (relational database) as a backend for configuration properties. You can enable this feature by adding `spring-jdbc` to the classpath and using the `jdbc` profile or by adding a bean of type `JdbcEnvironmentRepository`. If you include the right dependencies on the classpath (see the user guide for more details on that), Spring Boot configures a data source.

You can disable autoconfiguration for `JdbcEnvironmentRepository` by setting the `spring.cloud.config.server.jdbc.enabled` property to `false`.

The database needs to have a table called `PROPERTIES` with columns called `APPLICATION`, `PROFILE`, and `LABEL` (with the usual `Environment` meaning), plus `KEY` and `VALUE` for the key and value pairs in `Properties` style. All fields are of type String in Java, so you can make them `VARCHAR` of whatever length you need. Property values behave in the same way as they would if they came from Spring Boot properties files named `{application}-{profile}.properties`, including all the encryption and decryption, which will be applied as post-processing steps (that is, not in the repository implementation directly).

44.1.8. Redis Backend

Spring Cloud Config Server supports Redis as a backend for configuration properties. You can enable this feature by adding a dependency to Spring Data Redis.

pom.xml

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

The following configuration uses Spring Data `RedisTemplate` to access a Redis. We can use `spring.redis.*` properties to override default connection settings.

```
spring:
  profiles:
    active: redis
  redis:
    host: redis
    port: 16379
```

The properties should be stored as fields in a hash. The name of hash should be the same as `spring.application.name` property or conjunction of `spring.application.name` and `spring.profiles.active[n]`.

```
HMSET sample-app server.port "8100" sample.topic.name "test" test.property1
"property1"
```

After executing the command visible above a hash should contain the following keys with values:

```
HGETALL sample-app
{
  "server.port": "8100",
  "sample.topic.name": "test",
  "test.property1": "property1"
}
```

When no profile is specified `default` will be used.

44.1.9. AWS S3 Backend

Spring Cloud Config Server supports AWS S3 as a backend for configuration properties. You can enable this feature by adding a dependency to the AWS Java SDK For Amazon S3.

pom.xml

```
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-java-sdk-s3</artifactId>
    </dependency>
</dependencies>
```

The following configuration uses the AWS S3 client to access configuration files. We can use `spring.awss3.*` properties to select the bucket where your configuration is stored.

```
spring:
  profiles:
    active: awss3
  cloud:
    config:
      server:
        awss3:
          region: us-east-1
          bucket: bucket1
```

It is also possible to specify an AWS URL to override the standard endpoint of your S3 service with `spring.awss3.endpoint`. This allows support for beta regions of S3, and other S3 compatible storage APIs.

Credentials are found using the Default AWS Credential Provider Chain. Versioned and encrypted buckets are supported without further configuration.

Configuration files are stored in your bucket as `{application}-{profile}.properties`, `{application}-{profile}.yml` or `{application}-{profile}.json`. An optional label can be provided to specify a directory path to the file.

When no profile is specified `default` will be used.

44.1.10. CredHub Backend

Spring Cloud Config Server supports CredHub as a backend for configuration properties. You can enable this feature by adding a dependency to Spring CredHub.

pom.xml

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

The following configuration uses mutual TLS to access a CredHub:

```
spring:
  profiles:
    active: credhub
  cloud:
    config:
      server:
        credhub:
          url: https://credhub:8844
```

The properties should be stored as JSON, such as:

```
credhub set --name "/demo-app/default/master/toggles" --type=json
value: {"toggle.button": "blue", "toggle.link": "red"}
```

```
credhub set --name "/demo-app/default/master/abs" --type=json
value: {"marketing.enabled": true, "external.enabled": false}
```

All client applications with the name `spring.cloud.config.name=demo-app` will have the following properties available to them:

```
{
    toggle.button: "blue",
    toggle.link: "red",
    marketing.enabled: true,
    external.enabled: false
}
```

When no profile is specified `default` will be used and when no label is specified `master` will be used as a default value. NOTE: Values added to `application` will be shared by all the applications.

OAuth 2.0

You can authenticate with OAuth 2.0 using UAA as a provider.

pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-oauth2-client</artifactId>
    </dependency>
</dependencies>
```

The following configuration uses OAuth 2.0 and UAA to access a CredHub:

```
spring:
  profiles:
    active: credhub
  cloud:
    config:
      server:
        credhub:
          url: https://credhub:8844
          oauth2:
            registration-id: credhub-client
  security:
    oauth2:
      client:
        registration:
          credhub-client:
            provider: uaa
            client-id: credhub_config_server
            client-secret: asecret
            authorization-grant-type: client_credentials
        provider:
          uaa:
            token-uri: https://uaa:8443/oauth/token
```

The used UAA client-id should have `credhub.read` as scope.

44.1.11. Composite Environment Repositories

In some scenarios, you may wish to pull configuration data from multiple environment repositories. To do so, you can enable the `composite` profile in your configuration server's application properties or YAML file. If, for example, you want to pull configuration data from a Subversion repository as well as two Git repositories, you can set the following properties for your configuration server:

```yaml
spring:
  profiles:
    active: composite
  cloud:
    config:
      server:
        composite:
        -
          type: svn
          uri: file:///path/to/svn/repo
        -
          type: git
          uri: file:///path/to/rex/git/repo
        -
          type: git
          uri: file:///path/to/walter/git/repo
```

Using this configuration, precedence is determined by the order in which repositories are listed under the `composite` key. In the above example, the Subversion repository is listed first, so a value found in the Subversion repository will override values found for the same property in one of the Git repositories. A value found in the `rex` Git repository will be used before a value found for the same property in the `walter` Git repository.

If you want to pull configuration data only from repositories that are each of distinct types, you can enable the corresponding profiles, rather than the `composite` profile, in your configuration server's application properties or YAML file. If, for example, you want to pull configuration data from a single Git repository and a single HashiCorp Vault server, you can set the following properties for your configuration server:

```yaml
spring:
  profiles:
    active: git, vault
  cloud:
    config:
      server:
        git:
          uri: file:///path/to/git/repo
          order: 2
        vault:
          host: 127.0.0.1
          port: 8200
          order: 1
```

Using this configuration, precedence can be determined by an `order` property. You can use the `order` property to specify the priority order for all your repositories. The lower the numerical value of the `order` property, the higher priority it has. The priority order of a repository helps resolve any potential conflicts between repositories that contain values for the same properties.

If your composite environment includes a Vault server as in the previous example, you must include a Vault token in every request made to the configuration server. See Vault Backend.

Any type of failure when retrieving values from an environment repository results in a failure for the entire composite environment.

When using a composite environment, it is important that all repositories contain the same labels. If you have an environment similar to those in the preceding examples and you request configuration data with the `master` label but the Subversion repository does not contain a branch called `master`, the entire request fails.

Custom Composite Environment Repositories

In addition to using one of the environment repositories from Spring Cloud, you can also provide your own `EnvironmentRepository` bean to be included as part of a composite environment. To do so, your bean must implement the `EnvironmentRepository` interface. If you want to control the priority of your custom `EnvironmentRepository` within the composite environment, you should also implement the `Ordered` interface and override the `getOrdered` method. If you do not implement the `Ordered` interface, your `EnvironmentRepository` is given the lowest priority.

44.1.12. Property Overrides

The Config Server has an “overrides” feature that lets the operator provide configuration properties to all applications. The overridden properties cannot be accidentally changed by the application with the normal Spring Boot hooks. To declare overrides, add a map of name-value pairs to `spring.cloud.config.server.overrides`, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        overrides:
          foo: bar
```

The preceding examples causes all applications that are config clients to read `foo=bar`, independent of their own configuration.

A configuration system cannot force an application to use configuration data in any particular way. Consequently, overrides are not enforceable. However, they do provide useful default behavior for Spring Cloud Config clients.

Normally, Spring environment placeholders with `${}` can be escaped (and resolved on the client) by using backslash (`\`) to escape the `$` or the `{`. For example, `\${app.foo:bar}` resolves to `bar`, unless the app provides its own `app.foo`.

In YAML, you do not need to escape the backslash itself. However, in properties files, you do need to escape the backslash, when you configure the overrides on the server.

You can change the priority of all overrides in the client to be more like default values, letting applications supply their own values in environment variables or System properties, by setting the `spring.cloud.config.overrideNone=true` flag (the default is false) in the remote repository.

44.2. Health Indicator

Config Server comes with a Health Indicator that checks whether the configured `EnvironmentRepository` is working. By default, it asks the `EnvironmentRepository` for an application named `app`, the `default` profile, and the default label provided by the `EnvironmentRepository` implementation.

You can configure the Health Indicator to check more applications along with custom profiles and custom labels, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        health:
          repositories:
            myservice:
              label: mylabel
            myservice-dev:
              name: myservice
              profiles: development
```

You can disable the Health Indicator by setting `spring.cloud.config.server.health.enabled=false`.

44.3. Security

You can secure your Config Server in any way that makes sense to you (from physical network security to OAuth2 bearer tokens), because Spring Security and Spring Boot offer support for many security arrangements.

To use the default Spring Boot-configured HTTP Basic security, include Spring Security on the classpath (for example, through `spring-boot-starter-security`). The default is a username of `user` and a randomly generated password. A random password is not useful in practice, so we recommend you configure the password (by setting `spring.security.user.password`) and encrypt it (see below for instructions on how to do that).

44.4. Encryption and Decryption

To use the encryption and decryption features you need the full-strength JCE installed in your JVM (it is not included by default). You can download the “Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files” from Oracle and follow the installation instructions (essentially, you need to replace the two policy files in the JRE lib/security directory with the ones that you downloaded).

If the remote property sources contain encrypted content (values starting with `{cipher}`), they are decrypted before sending to clients over HTTP. The main advantage of this setup is that the property values need not be in plain text when they are “at rest” (for example, in a git repository). If a value cannot be decrypted, it is removed from the property source and an additional property is added with the same key but prefixed with `invalid` and a value that means “not applicable” (usually `<n/a>`). This is largely to prevent cipher text being used as a password and accidentally leaking.

If you set up a remote config repository for config client applications, it might contain an `application.yml` similar to the following:

application.yml

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
```

Encrypted values in a .properties file must not be wrapped in quotes. Otherwise, the value is not decrypted. The following example shows values that would work:

application.properties

```
spring.datasource.username: dbuser
spring.datasource.password: {cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ
```

You can safely push this plain text to a shared git repository, and the secret password remains protected.

The server also exposes `/encrypt` and `/decrypt` endpoints (on the assumption that these are secured and only accessed by authorized agents). If you edit a remote config file, you can use the Config Server to encrypt values by POSTing to the `/encrypt` endpoint, as shown in the following example:

```
$ curl localhost:8888/encrypt -s -d mysecret
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
```

If you are testing with curl, then use `--data-urlencode` (instead of `-d`) and prefix the value to encrypt with `=` (curl requires this) or set an explicit `Content-Type: text/plain` to make sure curl encodes the data correctly when there are special characters ('+' is particularly tricky).

Be sure not to include any of the curl command statistics in the encrypted value, this is why the examples use the `-s` option to silence them. Outputting the value to a file can help avoid this problem.

The inverse operation is also available through `/decrypt` (provided the server is configured with a symmetric key or a full key pair), as shown in the following example:

```
$ curl localhost:8888/decrypt -s -d
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

Take the encrypted value and add the `{cipher}` prefix before you put it in the YAML or properties file and before you commit and push it to a remote (potentially insecure) store.

The `/encrypt` and `/decrypt` endpoints also both accept paths in the form of `/*/{application}/{profiles}`, which can be used to control cryptography on a per-application (name) and per-profile basis when clients call into the main environment resource.

To control the cryptography in this granular way, you must also provide a `@Bean` of type `TextEncryptorLocator` that creates a different encryptor per name and profiles. The one that is provided by default does not do so (all encryptions use the same key).

The `spring` command line client (with Spring Cloud CLI extensions installed) can also be used to encrypt and decrypt, as shown in the following example:

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

To use a key in a file (such as an RSA public key for encryption), prepend the key value with "@" and provide the file path, as shown in the following example:

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

The `--key` argument is mandatory (despite having a `--` prefix).

44.5. Key Management

The Config Server can use a symmetric (shared) key or an asymmetric one (RSA key pair). The asymmetric choice is superior in terms of security, but it is often more convenient to use a symmetric key since it is a single property value to configure in the `bootstrap.properties`.

To configure a symmetric key, you need to set `encrypt.key` to a secret String (or use the `ENCRYPT_KEY` environment variable to keep it out of plain-text configuration files).

You cannot configure an asymmetric key using `encrypt.key`.

To configure an asymmetric key use a keystore (e.g. as created by the `keytool` utility that comes with the JDK). The keystore properties are `encrypt.keyStore.*` with `*` equal to

Property	Description
`encrypt.keyStore.location`	Contains a `Resource` location
`encrypt.keyStore.password`	Holds the password that unlocks the keystore
`encrypt.keyStore.alias`	Identifies which key in the store to use
`encrypt.keyStore.type`	The type of KeyStore to create. Defaults to `jks`.

The encryption is done with the public key, and a private key is needed for decryption. Thus, in principle, you can configure only the public key in the server if you want to only encrypt (and are prepared to decrypt the values yourself locally with the private key). In practice, you might not want to do decrypt locally, because it spreads the key management process around all the clients, instead of concentrating it in the server. On the other hand, it can be a useful option if your config server is relatively insecure and only a handful of clients need the encrypted properties.

44.6. Creating a Key Store for Testing

To create a keystore for testing, you can use a command resembling the following:

```
$ keytool -genkeypair -alias mytestkey -keyalg RSA \
  -dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \
  -keypass changeme -keystore server.jks -storepass letmein
```

When using JDK 11 or above you may get the following warning when using the command above. In this case you probably want to make sure the `keypass` and `storepass` values match.

```
Warning:  Different store and key passwords not supported for PKCS12 KeyStores.
Ignoring user-specified -keypass value.
```

Put the `server.jks` file in the classpath (for instance) and then, in your `bootstrap.yml`, for the Config Server, create the following settings:

```
encrypt:
  keyStore:
    location: classpath:/server.jks
    password: letmein
    alias: mytestkey
    secret: changeme
```

44.7. Using Multiple Keys and Key Rotation

In addition to the `{cipher}` prefix in encrypted property values, the Config Server looks for zero or more `{name:value}` prefixes before the start of the (Base64 encoded) cipher text. The keys are passed to a `TextEncryptorLocator`, which can do whatever logic it needs to locate a `TextEncryptor` for the cipher. If you have configured a keystore (`encrypt.keystore.location`), the default locator looks for keys with aliases supplied by the `key` prefix, with a cipher text like resembling the following:

```
foo:
  bar: `{cipher}{key:testkey}...`
```

The locator looks for a key named "testkey". A secret can also be supplied by using a `{secret:…}` value in the prefix. However, if it is not supplied, the default is to use the keystore password (which is what you get when you build a keystore and do not specify a secret). If you do supply a secret, you should also encrypt the secret using a custom `SecretLocator`.

When the keys are being used only to encrypt a few bytes of configuration data (that is, they are not being used elsewhere), key rotation is hardly ever necessary on cryptographic grounds. However, you might occasionally need to change the keys (for example, in the event of a security breach). In that case, all the clients would need to change their source config files (for example, in git) and use a new `{key:…}` prefix in all the ciphers. Note that the clients need to first check that the key alias is available in the Config Server keystore.

If you want to let the Config Server handle all encryption as well as decryption, the `{name:value}` prefixes can also be added as plain text posted to the `/encrypt` endpoint, .

44.8. Serving Encrypted Properties

Sometimes you want the clients to decrypt the configuration locally, instead of doing it in the server. In that case, if you provide the `encrypt.*` configuration to locate a key, you can still have `/encrypt` and `/decrypt` endpoints, but you need to explicitly switch off the decryption of outgoing properties by placing `spring.cloud.config.server.encrypt.enabled=false` in `bootstrap.[yml|properties]`. If you do not care about the endpoints, it should work if you do not configure either the key or the enabled flag.

Chapter 45. Serving Alternative Formats

The default JSON format from the environment endpoints is perfect for consumption by Spring applications, because it maps directly onto the `Environment` abstraction. If you prefer, you can consume the same data as YAML or Java properties by adding a suffix (".yml", ".yaml" or ".properties") to the resource path. This can be useful for consumption by applications that do not care about the structure of the JSON endpoints or the extra metadata they provide (for example, an application that is not using Spring might benefit from the simplicity of this approach).

The YAML and properties representations have an additional flag (provided as a boolean query parameter called `resolvePlaceholders`) to signal that placeholders in the source documents (in the standard Spring `${…}` form) should be resolved in the output before rendering, where possible. This is a useful feature for consumers that do not know about the Spring placeholder conventions.

There are limitations in using the YAML or properties formats, mainly in relation to the loss of metadata. For example, the JSON is structured as an ordered list of property sources, with names that correlate with the source. The YAML and properties forms are coalesced into a single map, even if the origin of the values has multiple sources, and the names of the original source files are lost. Also, the YAML representation is not necessarily a faithful representation of the YAML source in a backing repository either. It is constructed from a list of flat property sources, and assumptions have to be made about the form of the keys.

Chapter 46. Serving Plain Text

Instead of using the `Environment` abstraction (or one of the alternative representations of it in YAML or properties format), your applications might need generic plain-text configuration files that are tailored to their environment. The Config Server provides these through an additional endpoint at `/{application}/{profile}/{label}/{path}`, where `application`, `profile`, and `label` have the same meaning as the regular environment endpoint, but `path` is a path to a file name (such as `log.xml`). The source files for this endpoint are located in the same way as for the environment endpoints. The same search path is used for properties and YAML files. However, instead of aggregating all matching resources, only the first one to match is returned.

After a resource is located, placeholders in the normal format (`${…}`) are resolved by using the effective `Environment` for the supplied application name, profile, and label. In this way, the resource endpoint is tightly integrated with the environment endpoints.

As with the source files for environment configuration, the `profile` is used to resolve the file name. So, if you want a profile-specific file, `/*/development/*/logback.xml` can be resolved by a file called `logback-development.xml` (in preference to `logback.xml`).

If you do not want to supply the `label` and let the server use the default label, you can supply a `useDefaultLabel` request parameter. Consequently, the preceding example for the `default` profile could be `/sample/default/nginx.conf?useDefaultLabel`.

At present, Spring Cloud Config can serve plaintext for git, SVN, native backends, and AWS S3. The support for git, SVN, and native backends is identical. AWS S3 works a bit differently. The following sections show how each one works:

- Git, SVN, and Native Backends
- AWS S3

46.1. Git, SVN, and Native Backends

Consider the following example for a GIT or SVN repository or a native backend:

```
application.yml
nginx.conf
```

The `nginx.conf` might resemble the following listing:

```
server {
    listen              80;
    server_name         ${nginx.server.name};
}
```

`application.yml` might resemble the following listing:

```
nginx:
  server:
    name: example.com
---
spring:
  profiles: development
nginx:
  server:
    name: develop.com
```

The `/sample/default/master/nginx.conf` resource might be as follows:

```
server {
    listen              80;
    server_name         example.com;
}
```

`/sample/development/master/nginx.conf` might be as follows:

```
server {
    listen              80;
    server_name         develop.com;
}
```

46.2. AWS S3

To enable serving plain text for AWS s3, the Config Server application needs to include a dependency on Spring Cloud AWS. For details on how to set up that dependency, see the Spring Cloud AWS Reference Guide. Then you need to configure Spring Cloud AWS, as described in the Spring Cloud AWS Reference Guide.

46.3. Decrypting Plain Text

By default, encrypted values in plain text files are not decrypted. In order to enable decryption for plain text files, set `spring.cloud.config.server.encrypt.enabled=true` and `spring.cloud.config.server.encrypt.plainTextEncrypt=true` in `bootstrap.[yml|properties]`

Decrypting plain text files is only supported for YAML, JSON, and properties file extensions.

If this feature is enabled, and an unsupported file extention is requested, any encrypted values in the file will not be decrypted.

Chapter 47. Embedding the Config Server

The Config Server runs best as a standalone application. However, if need be, you can embed it in another application. To do so, use the `@EnableConfigServer` annotation. An optional property named `spring.cloud.config.server.bootstrap` can be useful in this case. It is a flag to indicate whether the server should configure itself from its own remote repository. By default, the flag is off, because it can delay startup. However, when embedded in another application, it makes sense to initialize the same way as any other application. When setting `spring.cloud.config.server.bootstrap` to `true` you must also use a composite environment repository configuration. For example

```
spring:
  application:
    name: configserver
  profiles:
    active: composite
  cloud:
    config:
      server:
        composite:
          - type: native
            search-locations: ${HOME}/Desktop/config
        bootstrap: true
```

If you use the bootstrap flag, the config server needs to have its name and repository URI configured in `bootstrap.yml`.

To change the location of the server endpoints, you can (optionally) set `spring.cloud.config.server.prefix` (for example, `/config`), to serve the resources under a prefix. The prefix should start but not end with a `/`. It is applied to the `@RequestMappings` in the Config Server (that is, underneath the Spring Boot `server.servletPath` and `server.contextPath` prefixes).

If you want to read the configuration for an application directly from the backend repository (instead of from the config server), you basically want an embedded config server with no endpoints. You can switch off the endpoints entirely by not using the `@EnableConfigServer` annotation (set `spring.cloud.config.server.bootstrap=true`).

Chapter 48. Push Notifications and Spring Cloud Bus

Many source code repository providers (such as Github, Gitlab, Gitea, Gitee, Gogs, or Bitbucket) notify you of changes in a repository through a webhook. You can configure the webhook through the provider's user interface as a URL and a set of events in which you are interested. For instance, Github uses a POST to the webhook with a JSON body containing a list of commits and a header (`X-Github-Event`) set to `push`. If you add a dependency on the `spring-cloud-config-monitor` library and activate the Spring Cloud Bus in your Config Server, then a `/monitor` endpoint is enabled.

When the webhook is activated, the Config Server sends a `RefreshRemoteApplicationEvent` targeted at the applications it thinks might have changed. The change detection can be strategized. However, by default, it looks for changes in files that match the application name (for example, `foo.properties` is targeted at the `foo` application, while `application.properties` is targeted at all applications). The strategy to use when you want to override the behavior is `PropertyPathNotificationExtractor`, which accepts the request headers and body as parameters and returns a list of file paths that changed.

The default configuration works out of the box with Github, Gitlab, Gitea, Gitee, Gogs or Bitbucket. In addition to the JSON notifications from Github, Gitlab, Gitee, or Bitbucket, you can trigger a change notification by POSTing to `/monitor` with form-encoded body parameters in the pattern of `path={application}`. Doing so broadcasts to applications matching the `{application}` pattern (which can contain wildcards).

The `RefreshRemoteApplicationEvent` is transmitted only if the `spring-cloud-bus` is activated in both the Config Server and in the client application.

The default configuration also detects filesystem changes in local git repositories. In that case, the webhook is not used. However, as soon as you edit a config file, a refresh is broadcast.

Chapter 49. Spring Cloud Config Client

A Spring Boot application can take immediate advantage of the Spring Config Server (or other external property sources provided by the application developer). It also picks up some additional useful features related to `Environment` change events.

49.1. Config First Bootstrap

The default behavior for any application that has the Spring Cloud Config Client on the classpath is as follows: When a config client starts, it binds to the Config Server (through the `spring.cloud.config.uri` bootstrap configuration property) and initializes Spring `Environment` with remote property sources.

The net result of this behavior is that all client applications that want to consume the Config Server need a `bootstrap.yml` (or an environment variable) with the server address set in `spring.cloud.config.uri` (it defaults to "http://localhost:8888").

49.2. Discovery First Bootstrap

If you use a `DiscoveryClient` implementation, such as Spring Cloud Netflix and Eureka Service Discovery or Spring Cloud Consul, you can have the Config Server register with the Discovery Service. However, in the default “Config First” mode, clients cannot take advantage of the registration.

If you prefer to use `DiscoveryClient` to locate the Config Server, you can do so by setting `spring.cloud.config.discovery.enabled=true` (the default is `false`). The net result of doing so is that client applications all need a `bootstrap.yml` (or an environment variable) with the appropriate discovery configuration. For example, with Spring Cloud Netflix, you need to define the Eureka server address (for example, in `eureka.client.serviceUrl.defaultZone`). The price for using this option is an extra network round trip on startup, to locate the service registration. The benefit is that, as long as the Discovery Service is a fixed point, the Config Server can change its coordinates. The default service ID is `configserver`, but you can change that on the client by setting `spring.cloud.config.discovery.serviceId` (and on the server, in the usual way for a service, such as by setting `spring.application.name`).

The discovery client implementations all support some kind of metadata map (for example, we have `eureka.instance.metadataMap` for Eureka). Some additional properties of the Config Server may need to be configured in its service registration metadata so that clients can connect correctly. If the Config Server is secured with HTTP Basic, you can configure the credentials as `user` and `password`. Also, if the Config Server has a context path, you can set `configPath`. For example, the following YAML file is for a Config Server that is a Eureka client:

bootstrap.yml

```
eureka:
  instance:
    ...
    metadataMap:
      user: osufhalskjrtl
      password: lviuhlszvaorhvlo5847
      configPath: /config
```

49.3. Config Client Fail Fast

In some cases, you may want to fail startup of a service if it cannot connect to the Config Server. If this is the desired behavior, set the bootstrap configuration property `spring.cloud.config.fail-fast=true` to make the client halt with an Exception.

49.4. Config Client Retry

If you expect that the config server may occasionally be unavailable when your application starts, you can make it keep trying after a failure. First, you need to set `spring.cloud.config.fail-fast=true`. Then you need to add `spring-retry` and `spring-boot-starter-aop` to your classpath. The default behavior is to retry six times with an initial backoff interval of 1000ms and an exponential multiplier of 1.1 for subsequent backoffs. You can configure these properties (and others) by setting the `spring.cloud.config.retry.*` configuration properties.

To take full control of the retry behavior, add a `@Bean` of type `RetryOperationsInterceptor` with an ID of `configServerRetryInterceptor`. Spring Retry has a `RetryInterceptorBuilder` that supports creating one.

49.5. Locating Remote Configuration Resources

The Config Service serves property sources from `/{application}/{profile}/{label}`, where the default bindings in the client app are as follows:

- "name" = `${spring.application.name}`
- "profile" = `${spring.profiles.active}` (actually `Environment.getActiveProfiles()`)
- "label" = "master"

When setting the property `${spring.application.name}` do not prefix your app name with the reserved word `application-` to prevent issues resolving the correct property source.

You can override all of them by setting `spring.cloud.config.*` (where `*` is `name`, `profile` or `label`). The `label` is useful for rolling back to previous versions of configuration. With the default Config Server implementation, it can be a git label, branch name, or commit ID. Label can also be provided as a comma-separated list. In that case, the items in the list are tried one by one until one succeeds. This

behavior can be useful when working on a feature branch. For instance, you might want to align the config label with your branch but make it optional (in that case, use `spring.cloud.config.label=myfeature,develop`).

49.6. Specifying Multiple Urls for the Config Server

To ensure high availability when you have multiple instances of Config Server deployed and expect one or more instances to be unavailable from time to time, you can either specify multiple URLs (as a comma-separated list under the `spring.cloud.config.uri` property) or have all your instances register in a Service Registry like Eureka (if using Discovery-First Bootstrap mode). Note that doing so ensures high availability only when the Config Server is not running (that is, when the application has exited) or when a connection timeout has occurred. For example, if the Config Server returns a 500 (Internal Server Error) response or the Config Client receives a 401 from the Config Server (due to bad credentials or other causes), the Config Client does not try to fetch properties from other URLs. An error of that kind indicates a user issue rather than an availability problem.

If you use HTTP basic security on your Config Server, it is currently possible to support per-Config Server auth credentials only if you embed the credentials in each URL you specify under the `spring.cloud.config.uri` property. If you use any other kind of security mechanism, you cannot (currently) support per-Config Server authentication and authorization.

49.7. Configuring Timeouts

If you want to configure timeout thresholds:

- Read timeouts can be configured by using the property `spring.cloud.config.request-read-timeout`.
- Connection timeouts can be configured by using the property `spring.cloud.config.request-connect-timeout`.

49.8. Security

If you use HTTP Basic security on the server, clients need to know the password (and username if it is not the default). You can specify the username and password through the config server URI or via separate username and password properties, as shown in the following example:

bootstrap.yml

```
spring:
  cloud:
    config:
     uri: https://user:secret@myconfig.mycompany.com
```

The following example shows an alternate way to pass the same information:

bootstrap.yml

```
spring:
  cloud:
    config:
     uri: https://myconfig.mycompany.com
     username: user
     password: secret
```

The `spring.cloud.config.password` and `spring.cloud.config.username` values override anything that is provided in the URI.

If you deploy your apps on Cloud Foundry, the best way to provide the password is through service credentials (such as in the URI, since it does not need to be in a config file). The following example works locally and for a user-provided service on Cloud Foundry named `configserver`:

bootstrap.yml

```
spring:
  cloud:
    config:
     uri:
${vcap.services.configserver.credentials.uri:http://user:password@localhost:8888}
```

If config server requires client side TLS certificate, you can configure client side TLS certificate and trust store via properties, as shown in following example:

bootstrap.yml

```
spring:
  cloud:
    config:
      uri: https://myconfig.myconfig.com
      tls:
        enabled: true
        key-store: <path-of-key-store>
        key-store-type: PKCS12
        key-store-password: <key-store-password>
        key-password: <key-password>
        trust-store: <path-of-trust-store>
        trust-store-type: PKCS12
        trust-store-password: <trust-store-password>
```

The `spring.cloud.config.tls.enabled` needs to be true to enable config client side TLS. When `spring.cloud.config.tls.trust-store` is omitted, a JVM default trust store is used. The default value for `spring.cloud.config.tls.key-store-type` and `spring.cloud.config.tls.trust-store-type` is PKCS12. When password properties are omitted, empty password is assumed.

If you use another form of security, you might need to provide a `RestTemplate` to the

ConfigServicePropertySourceLocator (for example, by grabbing it in the bootstrap context and injecting it).

49.8.1. Health Indicator

The Config Client supplies a Spring Boot Health Indicator that attempts to load configuration from the Config Server. The health indicator can be disabled by setting health.config.enabled=false. The response is also cached for performance reasons. The default cache time to live is 5 minutes. To change that value, set the health.config.time-to-live property (in milliseconds).

49.8.2. Providing A Custom RestTemplate

In some cases, you might need to customize the requests made to the config server from the client. Typically, doing so involves passing special Authorization headers to authenticate requests to the server. To provide a custom RestTemplate:

1. Create a new configuration bean with an implementation of PropertySourceLocator, as shown in the following example:

CustomConfigServiceBootstrapConfiguration.java

```
@Configuration
public class CustomConfigServiceBootstrapConfiguration {
    @Bean
    public ConfigServicePropertySourceLocator configServicePropertySourceLocator() {
        ConfigClientProperties clientProperties = configClientProperties();
       ConfigServicePropertySourceLocator configServicePropertySourceLocator =  new
ConfigServicePropertySourceLocator(clientProperties);

configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties
));
        return configServicePropertySourceLocator;
    }
}
```

For a simplified approach to adding Authorization headers, the spring.cloud.config.headers.* property can be used instead.

1. In resources/META-INF, create a file called spring.factories and specify your custom configuration, as shown in the following example:

spring.factories

```
org.springframework.cloud.bootstrap.BootstrapConfiguration =
com.my.config.client.CustomConfigServiceBootstrapConfiguration
```

49.8.3. Vault

When using Vault as a backend to your config server, the client needs to supply a token for the server to retrieve values from Vault. This token can be provided within the client by setting `spring.cloud.config.token` in `bootstrap.yml`, as shown in the following example:

bootstrap.yml

```
spring:
  cloud:
    config:
      token: YourVaultToken
```

49.9. Nested Keys In Vault

Vault supports the ability to nest keys in a value stored in Vault, as shown in the following example:

`echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' | vault write secret/myapp -`

This command writes a JSON object to your Vault. To access these values in Spring, you would use the traditional dot(`.`) annotation, as shown in the following example

```
@Value("${appA.secret}")
String name = "World";
```

The preceding code would sets the value of the `name` variable to `appAsecret`.

Spring Cloud Consul

Hoxton.SR8

This project provides Consul integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with Consul based components. The patterns provided include Service Discovery, Control Bus and Configuration. Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon), Circuit Breaker (Hystrix) are provided by integration with Spring Cloud Netflix.

Chapter 50. Install Consul

Please see the installation documentation for instructions on how to install Consul.

Chapter 51. Consul Agent

A Consul Agent client must be available to all Spring Cloud Consul applications. By default, the Agent client is expected to be at `localhost:8500`. See the Agent documentation for specifics on how to start an Agent client and how to connect to a cluster of Consul Agent Servers. For development, after you have installed consul, you may start a Consul Agent using the following command:

```
./src/main/bash/local_run_consul.sh
```

This will start an agent in server mode on port 8500, with the ui available at localhost:8500

Chapter 52. Service Discovery with Consul

Service Discovery is one of the key tenets of a microservice based architecture. Trying to hand configure each client or some form of convention can be very difficult to do and can be very brittle. Consul provides Service Discovery services via an HTTP API and DNS. Spring Cloud Consul leverages the HTTP API for service registration and discovery. This does not prevent non-Spring Cloud applications from leveraging the DNS interface. Consul Agents servers are run in a cluster that communicates via a gossip protocol and uses the Raft consensus protocol.

52.1. How to activate

To activate Consul Service Discovery use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-consul-discovery`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

52.2. Registering with Consul

When a client registers with Consul, it provides meta-data about itself such as host and port, id, name and tags. An HTTP Check is created by default that Consul hits the `/health` endpoint every 10 seconds. If the health check fails, the service instance is marked as critical.

Example Consul client:

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

(i.e. utterly normal Spring Boot app). If the Consul client is located somewhere other than `localhost:8500`, the configuration is required to locate the client. Example:

application.yml

```
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
```

If you use Spring Cloud Consul Config, the above values will need to be placed in `bootstrap.yml` instead of `application.yml`.

The default service name, instance id and port, taken from the `Environment`, are `${spring.application.name}`, the Spring Context ID and `${server.port}` respectively.

To disable the Consul Discovery Client you can set `spring.cloud.consul.discovery.enabled` to `false`. Consul Discovery Client will also be disabled when `spring.cloud.discovery.enabled` is set to `false`.

To disable the service registration you can set `spring.cloud.consul.discovery.register` to `false`.

52.2.1. Registering Management as a Separate Service

When management server port is set to something different than the application port, by setting `management.server.port` property, management service will be registered as a separate service than the application service. For example:

application.yml

```
spring:
  application:
    name: myApp
management:
  server:
    port: 4452
```

Above configuration will register following 2 services:

- Application Service:

```
ID: myApp
Name: myApp
```

- Management Service:

```
ID: myApp-management
Name: myApp-management
```

Management service will inherit its `instanceId` and `serviceName` from the application service. For

example:

application.yml

```
spring:
  application:
    name: myApp
management:
  server:
    port: 4452
spring:
  cloud:
    consul:
      discovery:
        instance-id: custom-service-id
        serviceName: myprefix-${spring.application.name}
```

Above configuration will register following 2 services:

- Application Service:

```
ID: custom-service-id
Name: myprefix-myApp
```

- Management Service:

```
ID: custom-service-id-management
Name: myprefix-myApp-management
```

Further customization is possible via following properties:

```
/** Port to register the management service under (defaults to management port) */
spring.cloud.consul.discovery.management-port

/** Suffix to use when registering management service (defaults to "management" */
spring.cloud.consul.discovery.management-suffix

/** Tags to use when registering management service (defaults to "management" */
spring.cloud.consul.discovery.management-tags
```

52.3. HTTP Health Check

The health check for a Consul instance defaults to "/health", which is the default locations of a useful endpoint in a Spring Boot Actuator application. You need to change these, even for an Actuator application if you use a non-default context path or servlet path (e.g. `server.servletPath=/foo`) or management endpoint path (e.g. `management.server.servlet.context-`

`path=/admin`). The interval that Consul uses to check the health endpoint may also be configured. "10s" and "1m" represent 10 seconds and 1 minute respectively. Example:

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        healthCheckPath: ${management.server.servlet.context-path}/health
        healthCheckInterval: 15s
```

You can disable the health check by setting `management.health.consul.enabled=false`.

52.3.1. Metadata and Consul tags

Consul does not yet support metadata on services. Spring Cloud's `ServiceInstance` has a `Map<String, String> metadata` field. Spring Cloud Consul uses Consul tags to approximate metadata until Consul officially supports metadata. Tags with the form `key=value` will be split and used as a `Map` key and value respectively. Tags without the equal `=` sign, will be used as both the key and value.

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        tags: foo=bar, baz
```

The above configuration will result in a map with `foo→bar` and `baz→baz`.

Generated Metadata

The Consul Auto Registration will generate a few entries automatically.

Table 4. Auto Generated Metadata

Key	Value
'group'	Property `spring.cloud.consul.discovery.instance-group`. This values is only generated if `instance-group` is not empty.'
'secure'	True if property `spring.cloud.consul.discovery.scheme` equals 'https', otherwise false.

Key	Value
Property spring.cloud.consul.discovery.default-zone-metadata-name, defaults to 'zone'	Property spring.cloud.consul.discovery.instance-zone. This values is only generated if instance-zone is not empty.'

Official Consul Metadata

Consul added official support for a meta field that is a Map<String, String>. Spring Cloud Consul has added spring.cloud.consul.discovery.metadata and spring.cloud.consul.discovery.management-metadata properties to support it.

By default, the ServiceInstance.getMetadata() method from Spring Cloud Commons will continue to populated by parsing the spring.cloud.consul.discovery.tags property for backwards compatibility. To change this behaviour set spring.cloud.consul.discovery.tags-as-metadata=false and the metadata will be populated from spring.cloud.consul.discovery.metadata. In a future version, parsing the tags property will be removed.

52.3.2. Making the Consul Instance ID Unique

By default a consul instance is registered with an ID that is equal to its Spring Application Context ID. By default, the Spring Application Context ID is ${spring.application.name}:comma,separated,profiles:${server.port}. For most cases, this will allow multiple instances of one service to run on one machine. If further uniqueness is required, Using Spring Cloud you can override this by providing a unique identifier in spring.cloud.consul.discovery.instanceId. For example:

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        instanceId:
${spring.application.name}:${vcap.application.instance_id:${spring.application.instanc
e_id:${random.value}}}
```

With this metadata, and multiple service instances deployed on localhost, the random value will kick in there to make the instance unique. In Cloudfoundry the vcap.application.instance_id will be populated automatically in a Spring Boot application, so the random value will not be needed.

52.3.3. Applying Headers to Health Check Requests

Headers can be applied to health check requests. For example, if you're trying to register a Spring Cloud Config server that uses Vault Backend:

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        health-check-headers:
          X-Config-Token: 6442e58b-d1ea-182e-cfa5-cf9cddef0722
```

According to the HTTP standard, each header can have more than one values, in which case, an array can be supplied:

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        health-check-headers:
          X-Config-Token:
            - "6442e58b-d1ea-182e-cfa5-cf9cddef0722"
            - "Some other value"
```

52.4. Looking up services

52.4.1. Using Load-balancer

Spring Cloud has support for Feign (a REST client builder) and also Spring `RestTemplate` for looking up services using the logical service names/ids instead of physical URLs. Both Feign and the discovery-aware RestTemplate utilize Ribbon for client-side load balancing.

If you want to access service STORES using the RestTemplate simply declare:

```
@LoadBalanced
@Bean
public RestTemplate loadbalancedRestTemplate() {
     return new RestTemplate();
}
```

and use it like this (notice how we use the STORES service name/id from Consul instead of a fully qualified domainname):

```
@Autowired
RestTemplate restTemplate;

public String getFirstProduct() {
   return this.restTemplate.getForObject("https://STORES/products/1", String.class);
}
```

If you have Consul clusters in multiple datacenters and you want to access a service in another datacenter a service name/id alone is not enough. In that case you use property `spring.cloud.consul.discovery.datacenters.STORES=dc-west` where `STORES` is the service name/id and `dc-west` is the datacenter where the STORES service lives.

Spring Cloud now also offers support for Spring Cloud LoadBalancer.

As Spring Cloud Ribbon is now under maintenance, we suggest you set `spring.cloud.loadbalancer.ribbon.enabled` to `false`, so that `BlockingLoadBalancerClient` is used instead of `RibbonLoadBalancerClient`.

52.4.2. Using the DiscoveryClient

You can also use the `org.springframework.cloud.client.discovery.DiscoveryClient` which provides a simple API for discovery clients that is not specific to Netflix, e.g.

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
    }
    return null;
}
```

52.5. Consul Catalog Watch

The Consul Catalog Watch takes advantage of the ability of consul to watch services. The Catalog Watch makes a blocking Consul HTTP API call to determine if any services have changed. If there is new service data a Heartbeat Event is published.

To change the frequency of when the Config Watch is called change `spring.cloud.consul.config.discovery.catalog-services-watch-delay`. The default value is 1000, which is in milliseconds. The delay is the amount of time after the end of the previous invocation and the start of the next.

To disable the Catalog Watch set `spring.cloud.consul.discovery.catalogServicesWatch.enabled=false`.

The watch uses a Spring `TaskScheduler` to schedule the call to consul. By default it is a `ThreadPoolTaskScheduler` with a `poolSize` of 1. To change the `TaskScheduler`, create a bean of type `TaskScheduler` named with the `ConsulDiscoveryClientConfiguration.CATALOG_WATCH_TASK_SCHEDULER_NAME` constant.

Chapter 53. Distributed Configuration with Consul

Consul provides a Key/Value Store for storing configuration and other metadata. Spring Cloud Consul Config is an alternative to the Config Server and Client. Configuration is loaded into the Spring Environment during the special "bootstrap" phase. Configuration is stored in the `/config` folder by default. Multiple `PropertySource` instances are created based on the application's name and the active profiles that mimicks the Spring Cloud Config order of resolving properties. For example, an application with the name "testApp" and with the "dev" profile will have the following property sources created:

```
config/testApp,dev/
config/testApp/
config/application,dev/
config/application/
```

The most specific property source is at the top, with the least specific at the bottom. Properties in the `config/application` folder are applicable to all applications using consul for configuration. Properties in the `config/testApp` folder are only available to the instances of the service named "testApp".

Configuration is currently read on startup of the application. Sending a HTTP POST to `/refresh` will cause the configuration to be reloaded. Config Watch will also automatically detect changes and reload the application context.

53.1. How to activate

To get started with Consul Configuration use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-consul-config`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

This will enable auto-configuration that will setup Spring Cloud Consul Config.

53.2. Customizing

Consul Config may be customized using the following properties:

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        enabled: true
        prefix: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- `enabled` setting this value to "false" disables Consul Config
- `prefix` sets the base folder for configuration values
- `defaultContext` sets the folder name used by all applications
- `profileSeparator` sets the value of the separator used to separate the profile name in property sources with profiles

53.3. Config Watch

The Consul Config Watch takes advantage of the ability of consul to watch a key prefix. The Config Watch makes a blocking Consul HTTP API call to determine if any relevant configuration data has changed for the current application. If there is new configuration data a Refresh Event is published. This is equivalent to calling the `/refresh` actuator endpoint.

To change the frequency of when the Config Watch is called change `spring.cloud.consul.config.watch.delay`. The default value is 1000, which is in milliseconds. The delay is the amount of time after the end of the previous invocation and the start of the next.

To disable the Config Watch set `spring.cloud.consul.config.watch.enabled=false`.

The watch uses a Spring `TaskScheduler` to schedule the call to consul. By default it is a `ThreadPoolTaskScheduler` with a `poolSize` of 1. To change the `TaskScheduler`, create a bean of type `TaskScheduler` named with the `ConsulConfigAutoConfiguration.CONFIG_WATCH_TASK_SCHEDULER_NAME` constant.

53.4. YAML or Properties with Config

It may be more convenient to store a blob of properties in YAML or Properties format as opposed to individual key/value pairs. Set the `spring.cloud.consul.config.format` property to `YAML` or `PROPERTIES`. For example to use YAML:

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        format: YAML
```

YAML must be set in the appropriate `data` key in consul. Using the defaults above the keys would look like:

```
config/testApp,dev/data
config/testApp/data
config/application,dev/data
config/application/data
```

You could store a YAML document in any of the keys listed above.

You can change the data key using `spring.cloud.consul.config.data-key`.

53.5. git2consul with Config

git2consul is a Consul community project that loads files from a git repository to individual keys into Consul. By default the names of the keys are names of the files. YAML and Properties files are supported with file extensions of `.yml` and `.properties` respectively. Set the `spring.cloud.consul.config.format` property to `FILES`. For example:

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        format: FILES
```

Given the following keys in `/config`, the `development` profile and an application name of `foo`:

```
.gitignore
application.yml
bar.properties
foo-development.properties
foo-production.yml
foo.properties
master.ref
```

the following property sources would be created:

```
config/foo-development.properties
config/foo.properties
config/application.yml
```

The value of each key needs to be a properly formatted YAML or Properties file.

53.6. Fail Fast

It may be convenient in certain circumstances (like local development or certain test scenarios) to not fail if consul isn't available for configuration. Setting `spring.cloud.consul.config.failFast=false` in `bootstrap.yml` will cause the configuration module to log a warning rather than throw an exception. This will allow the application to continue startup normally.

Chapter 54. Consul Retry

If you expect that the consul agent may occasionally be unavailable when your app starts, you can ask it to keep trying after a failure. You need to add `spring-retry` and `spring-boot-starter-aop` to your classpath. The default behaviour is to retry 6 times with an initial backoff interval of 1000ms and an exponential multiplier of 1.1 for subsequent backoffs. You can configure these properties (and others) using `spring.cloud.consul.retry.*` configuration properties. This works with both Spring Cloud Consul Config and Discovery registration.

To take full control of the retry add a `@Bean` of type `RetryOperationsInterceptor` with id "consulRetryInterceptor". Spring Retry has a `RetryInterceptorBuilder` that makes it easy to create one.

Chapter 55. Spring Cloud Bus with Consul

55.1. How to activate

To get started with the Consul Bus use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-consul-bus`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

See the Spring Cloud Bus documentation for the available actuator endpoints and howto send custom messages.

Chapter 56. Circuit Breaker with Hystrix

Applications can use the Hystrix Circuit Breaker provided by the Spring Cloud Netflix project by including this starter in the projects pom.xml: `spring-cloud-starter-hystrix`. Hystrix doesn't depend on the Netflix Discovery Client. The `@EnableHystrix` annotation should be placed on a configuration class (usually the main class). Then methods can be annotated with `@HystrixCommand` to be protected by a circuit breaker. See the documentation for more details.

Chapter 57. Hystrix metrics aggregation with Turbine and Consul

Turbine (provided by the Spring Cloud Netflix project), aggregates multiple instances Hystrix metrics streams, so the dashboard can display an aggregate view. Turbine uses the `DiscoveryClient` interface to lookup relevant instances. To use Turbine with Spring Cloud Consul, configure the Turbine application in a manner similar to the following examples:

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

Notice that the Turbine dependency is not a starter. The turbine starter includes support for Netflix Eureka.

application.yml

```
spring.application.name: turbine
applications: consulhystrixclient
turbine:
  aggregator:
    clusterConfig: ${applications}
  appConfig: ${applications}
```

The `clusterConfig` and `appConfig` sections must match, so it's useful to put the comma-separated list of service ID's into a separate configuration property.

Turbine.java

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

Chapter 58. Configuration Properties

To see the list of all Consul related configuration properties please check the Appendix page.

index.htmladoc

Spring Cloud Function

Mark Fisher, Dave Syer, Oleg Zhurakousky, Anshul Mehra

3.0.10.RELEASE

Chapter 59. Introduction

Spring Cloud Function is a project with the following high-level goals:

- Promote the implementation of business logic via functions.
- Decouple the development lifecycle of business logic from any specific runtime target so that the same code can run as a web endpoint, a stream processor, or a task.
- Support a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS).
- Enable Spring Boot features (auto-configuration, dependency injection, metrics) on serverless providers.

It abstracts away all of the transport details and infrastructure, allowing the developer to keep all the familiar tools and processes, and focus firmly on business logic.

Here's a complete, executable, testable Spring Boot application (implementing a simple string manipulation):

```
@SpringBootApplication
public class Application {

  @Bean
  public Function<Flux<String>, Flux<String>> uppercase() {
    return flux -> flux.map(value -> value.toUpperCase());
  }

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

It's just a Spring Boot application, so it can be built, run and tested, locally and in a CI build, the same way as any other Spring Boot application. The `Function` is from `java.util` and `Flux` is a Reactive Streams `Publisher` from Project Reactor. The function can be accessed over HTTP or messaging.

Spring Cloud Function has 4 main features:

In the nutshell Spring Cloud Function provides the following features: 1. Wrappers for `@Beans` of type `Function`, `Consumer` and `Supplier`, exposing them to the outside world as either HTTP endpoints and/or message stream listeners/publishers with RabbitMQ, Kafka etc.

- *Choice of programming styles - reactive, imperative or hybrid.*
- *Function composition and adaptation (e.g., composing imperative functions with reactive).*
- *Support for reactive function with multiple inputs and outputs allowing merging, joining and other complex streaming operation to be handled by functions.*

- *Transparent type conversion of inputs and outputs.*
- *Packaging functions for deployments, specific to the target platform (e.g., Project Riff, AWS Lambda and more)*
- *Adapters to expose function to the outside world as HTTP endpoints etc.*
- *Deploying a JAR file containing such an application context with an isolated classloader, so that you can pack them together in a single JVM.*
- *Compiling strings which are Java function bodies into bytecode, and then turning them into* `@Beans` *that can be wrapped as above.*
- *Adapters for AWS Lambda, Azure, Google Cloud Functions, Apache OpenWhisk and possibly other "serverless" service providers.*

Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at github.

Chapter 60. Getting Started

Build from the command line (and "install" the samples):

```
$ ./mvnw clean install
```

(If you like to YOLO add `-DskipTests`.)

Run one of the samples, e.g.

```
$ java -jar spring-cloud-function-samples/function-sample/target/*.jar
```

This runs the app and exposes its functions over HTTP, so you can convert a string to uppercase, like this:

```
$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d Hello
HELLO
```

You can convert multiple strings (a `Flux<String>`) by separating them with new lines

```
$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d 'Hello
> World'
HELLOWORLD
```

(You can use `^J` in a terminal to insert a new line in a literal string like that.)

Chapter 61. Programming model

61.1. Function Catalog and Flexible Function Signatures

One of the main features of Spring Cloud Function is to adapt and support a range of type signatures for user-defined functions, while providing a consistent execution model. That's why all user defined functions are transformed into a canonical representation by `FunctionCatalog`.

While users don't normally have to care about the `FunctionCatalog` at all, it is useful to know what kind of functions are supported in user code.

It is also important to understand that Spring Cloud Function provides first class support for reactive API provided by Project Reactor allowing reactive primitives such as `Mono` and `Flux` to be used as types in user defined functions providing greater flexibility when choosing programming model for your function implementation. Reactive programming model also enables functional support for features that would be otherwise difficult to impossible to implement using imperative programming style. For more on this please read Function Arity section.

61.2. Java 8 function support

Spring Cloud Function embraces and builds on top of the 3 core functional interfaces defined by Java and available to us since Java 8.

- Supplier<O>
- Function<I, O>
- Consumer<I>

61.2.1. Supplier

Supplier can be *reactive* - `Supplier<Flux<T>>` or *imperative* - `Supplier<T>`. From the invocation standpoint this should make no difference to the implementor of such Supplier. However, when used within frameworks (e.g., Spring Cloud Stream), Suppliers, especially reactive, often used to represent the source of the stream, therefore they are invoked once to get the stream (e.g., Flux) to which consumers can subscribe to. In other words such suppliers represent an equivalent of an *infinite stream*. However, the same reactive suppliers can also represent *finite* stream(s) (e.g., result set on the polled JDBC data). In those cases such reactive suppliers must be hooked up to some polling mechanism of the underlying framework.

To assist with that Spring Cloud Function provides a marker annotation `org.springframework.cloud.function.context.PollableSupplier` to signal that such supplier produces a finite stream and may need to be polled again. That said, it is important to understand that Spring Cloud Function itself provides no behavior for this annotation.

In addition `PollableSupplier` annotation exposes a *splittable* attribute to signal that produced stream needs to be split (see Splitter EIP)

Here is the example:

```
@PollableSupplier(splittable = true)
public Supplier<Flux<String>> someSupplier() {
    return () -> {
        String v1 = String.valueOf(System.nanoTime());
        String v2 = String.valueOf(System.nanoTime());
        String v3 = String.valueOf(System.nanoTime());
        return Flux.just(v1, v2, v3);
    };
}
```

61.2.2. Function

Function can also be written in imperative or reactive way, yet unlike Supplier and Consumer there are no special considerations for the implementor other then understanding that when used within frameworks such as Spring Cloud Stream and others, reactive function is invoked only once to pass a reference to the stream (Flux or Mono) and imperative is invoked once per event.

61.2.3. Consumer

Consumer is a little bit special because it has a `void` return type, which implies blocking, at least potentially. Most likely you will not need to write `Consumer<Flux<?>>`, but if you do need to do that, remember to subscribe to the input flux.

61.3. Function Composition

Function Composition is a feature that allows one to compose several functions into one. The core support is based on function composition feature available with Function.andThen(..) support available since Java 8. However on top of it, we provide few additional features.

61.3.1. Declarative Function Composition

This feature allows you to provide composition instruction in a declarative way using `|` (pipe) or `,` (comma) delimiter when providing `spring.cloud.function.definition` property.

For example

```
--spring.cloud.function.definition=uppercase|reverse
```

Here we effectively provided a definition of a single function which itself is a composition of function `uppercase` and function `reverse`. In fact that is one of the reasons why the property name is *definition* and not *name*, since the definition of a function can be a composition of several named functions. And as mentioned you can use `,` instead of pipe (such as `…definition=uppercase,reverse`).

61.3.2. Composing non-Functions

Spring Cloud Function also supports composing Supplier with `Consumer` or `Function` as well as `Function` with `Consumer`. What's important here is to understand the end product of such definitions. Composing Supplier with Function still results in Supplier while composing Supplier with Consumer will effectively render Runnable. Following the same logic composing Function with Consumer will result in Consumer.

And of course you can't compose uncomposable such as Consumer and Function, Consumer and Supplier etc.

61.4. Function Routing

Since version 2.2 Spring Cloud Function provides routing feature allowing you to invoke a single function which acts as a router to an actual function you wish to invoke This feature is very useful in certain FAAS environments where maintaining configurations for several functions could be cumbersome or exposing more then one function is not possible.

The `RoutingFunction` is registered in *FunctionCatalog* under the name `functionRouter`. For simplicity and consistency you can also refer to `RoutingFunction.FUNCTION_NAME` constant.

This function has the following signature:

```
public class RoutingFunction implements Function<Object, Object> {
. . .
}
```

The routing instructions could be communicated in several ways;

Message Headers

If the input argument is of type `Message<?>`, you can communicate routing instruction by setting one of `spring.cloud.function.definition` or `spring.cloud.function.routing-expression` Message headers. For more static cases you can use `spring.cloud.function.definition` header which allows you to provide the name of a single function (e.g., `…definition=foo`) or a composition instruction (e.g., `…definition=foo|bar|baz`). For more dynamic cases you can use `spring.cloud.function.routing-expression` header which allows you to use Spring Expression Language (SpEL) and provide SpEL expression that should resolve into definition of a function (as described above).

SpEL evaluation context's root object is the actual input argument, so in he case of `Message<?>` you can construct expression that has access to both `payload` and `headers` (e.g., `spring.cloud.function.routing-expression=headers.function_name`).

In specific execution environments/models the adapters are responsible to translate and communicate `spring.cloud.function.definition` and/or `spring.cloud.function.routing-expression` via Message header. For example, when using *spring-cloud-function-web* you can provide `spring.cloud.function.definition` as an HTTP header and the framework will propagate it as well as other HTTP headers as Message headers.

Application Properties

Routing instruction can also be communicated via `spring.cloud.function.definition` or `spring.cloud.function.routing-expression` as application properties. The rules described in the previous section apply here as well. The only difference is you provide these instructions as application properties (e.g., `--spring.cloud.function.definition=foo`).

When dealing with reactive inputs (e.g., Publisher), routing instructions must only be provided via Function properties. This is due to the nature of the reactive functions which are invoked only once to pass a Publisher and the rest is handled by the reactor, hence we can not access and/or rely on the routing instructions communicated via individual values (e.g., Message).

61.5. Function Arity

There are times when a stream of data needs to be categorized and organized. For example, consider a classic big-data use case of dealing with unorganized data containing, let's say, 'orders' and 'invoices', and you want each to go into a separate data store. This is where function arity (functions with multiple inputs and outputs) support comes to play.

Let's look at an example of such a function (full implementation details are available here),

```
@Bean
public Function<Flux<Integer>, Tuple2<Flux<String>, Flux<String>>> organise() {
    return flux -> ...;
}
```

Given that Project Reactor is a core dependency of SCF, we are using its Tuple library. Tuples give us a unique advantage by communicating to us both *cardinality* and *type* information. Both are extremely important in the context of SCSt. Cardinality lets us know how many input and output bindings need to be created and bound to the corresponding inputs and outputs of a function. Awareness of the type information ensures proper type conversion.

Also, this is where the 'index' part of the naming convention for binding names comes into play, since, in this function, the two output binding names are `organise-out-0` and `organise-out-1`.

IMPORTANT: At the moment, function arity is **only** supported for reactive functions (`Function<TupleN<Flux<?>…>, TupleN<Flux<?>…>>`) centered on Complex event processing where evaluation and computation on confluence of events typically requires view into a stream of events rather than single event.

61.6. Type conversion (Content-Type negotiation)

Content-Type negotiation is one of the core features of Spring Cloud Function as it allows to not only transform the incoming data to the types declared by the function signature, but to do the same transformation during function composition making otherwise un-composable (by type)

functions composable.

To better understand the mechanics and the necessity behind content-type negotiation, we take a look at a very simple use case by using the following function as an example:

```
@Bean
public Function<Person, String> personFunction {..}
```

The function shown in the preceding example expects a `Person` object as an argument and produces a String type as an output. If such function is invoked with the type `Person`, than all works fine. But typically function plays a role of a handler for the incoming data which most often comes in the raw format such as `byte[]`, `JSON String` etc. In order for the framework to succeed in passing the incoming data as an argument to this function, it has to somehow transform the incoming data to a `Person` type.

Spring Cloud Function relies on two native to Spring mechanisms to accomplish that.

1. *MessageConverter* - to convert from incoming Message data to a type declared by the function.
2. *ConversionService* - to convert from incoming non-Message data to a type declared by the function.

This means that depending on the type of the raw data (Message or non-Message) Spring Cloud Function will apply one or the other mechanisms.

For most cases when dealing with functions that are invoked as part of some other request (e.g., HTTP, Messaging etc) the framework relies on `MessageConverters`, since such requests already converted to Spring `Message`. In other words, the framework locates and applies the appropriate `MessageConverter`. To accomplish that, the framework needs some instructions from the user. One of these instructions is already provided by the signature of the function itself (Person type). Consequently, in theory, that should be (and, in some cases, is) enough. However, for the majority of use cases, in order to select the appropriate `MessageConverter`, the framework needs an additional piece of information. That missing piece is `contentType` header.

Such header usually comes as part of the Message where it is injected by the corresponding adapter that created such Message in the first place. For example, HTTP POST request will have its content-type HTTP header copied to `contentType` header of the Message.

For cases when such header does not exist framework relies on the default content type as `application/json`.

61.6.1. Content Type versus Argument Type

As mentioned earlier, for the framework to select the appropriate `MessageConverter`, it requires argument type and, optionally, content type information. The logic for selecting the appropriate `MessageConverter` resides with the argument resolvers which trigger right before the invocation of the user-defined function (which is when the actual argument type is known to the framework). If the argument type does not match the type of the current payload, the framework delegates to the stack of the pre-configured `MessageConverters` to see if any one of them can convert the payload.

The combination of `contentType` and argument type is the mechanism by which framework determines if message can be converted to a target type by locating the appropriate `MessageConverter`. If no appropriate `MessageConverter` is found, an exception is thrown, which you can handle by adding a custom `MessageConverter` (see `User-defined Message Converters`).

Do not expect `Message` to be converted into some other type based only on the `contentType`. Remember that the `contentType` is complementary to the target type. It is a hint, which `MessageConverter` may or may not take into consideration.

61.6.2. Message Converters

`MessageConverters` define two methods:

```
Object fromMessage(Message<?> message, Class<?> targetClass);

Message<?> toMessage(Object payload, @Nullable MessageHeaders headers);
```

It is important to understand the contract of these methods and their usage, specifically in the context of Spring Cloud Stream.

The `fromMessage` method converts an incoming `Message` to an argument type. The payload of the `Message` could be any type, and it is up to the actual implementation of the `MessageConverter` to support multiple types.

61.6.3. Provided MessageConverters

As mentioned earlier, the framework already provides a stack of `MessageConverters` to handle most common use cases. The following list describes the provided `MessageConverters`, in order of precedence (the first `MessageConverter` that works is used):

1. `JsonMessageConverter`: Supports conversion of the payload of the `Message` to/from POJO for cases when `contentType` is `application/json` using Jackson or Gson libraries (DEFAULT).
2. `ByteArrayMessageConverter`: Supports conversion of the payload of the `Message` from `byte[]` to `byte[]` for cases when `contentType` is `application/octet-stream`. It is essentially a pass through and exists primarily for backward compatibility.
3. `StringMessageConverter`: Supports conversion of any type to a `String` when `contentType` is `text/plain`.

When no appropriate converter is found, the framework throws an exception. When that happens, you should check your code and configuration and ensure you did not miss anything (that is, ensure that you provided a `contentType` by using a binding or a header). However, most likely, you found some uncommon case (such as a custom `contentType` perhaps) and the current stack of provided `MessageConverters` does not know how to convert. If that is the case, you can add custom `MessageConverter`. See User-defined Message Converters.

61.6.4. User-defined Message Converters

Spring Cloud Function exposes a mechanism to define and register additional `MessageConverters`. To use it, implement `org.springframework.messaging.converter.MessageConverter`, configure it as a `@Bean`. It is then appended to the existing stack of `MessageConverter`s.

It is important to understand that custom `MessageConverter` implementations are added to the head of the existing stack. Consequently, custom `MessageConverter` implementations take precedence over the existing ones, which lets you override as well as add to the existing converters.

The following example shows how to create a message converter bean to support a new content type called `application/bar`:

```
@SpringBootApplication
public static class SinkApplication {

    ...

    @Bean
    public MessageConverter customMessageConverter() {
        return new MyCustomMessageConverter();
    }
}

public class MyCustomMessageConverter extends AbstractMessageConverter {

    public MyCustomMessageConverter() {
        super(new MimeType("application", "bar"));
    }

    @Override
    protected boolean supports(Class<?> clazz) {
        return (Bar.class.equals(clazz));
    }

    @Override
    protected Object convertFromInternal(Message<?> message, Class<?> targetClass,
Object conversionHint) {
        Object payload = message.getPayload();
        return (payload instanceof Bar ? payload : new Bar((byte[]) payload));
    }
}
```

61.7. Kotlin Lambda support

We also provide support for Kotlin lambdas (since v2.0). Consider the following:

```
@Bean
open fun kotlinSupplier(): () -> String {
    return  { "Hello from Kotlin" }
}

@Bean
open fun kotlinFunction(): (String) -> String {
    return  { it.toUpperCase() }
}

@Bean
open fun kotlinConsumer(): (String) -> Unit {
    return  { println(it) }
}
```

The above represents Kotlin lambdas configured as Spring beans. The signature of each maps to a Java equivalent of `Supplier`, `Function` and `Consumer`, and thus supported/recognized signatures by the framework. While mechanics of Kotlin-to-Java mapping are outside of the scope of this documentation, it is important to understand that the same rules for signature transformation outlined in "Java 8 function support" section are applied here as well.

To enable Kotlin support all you need is to add `spring-cloud-function-kotlin` module to your classpath which contains the appropriate autoconfiguration and supporting classes.

61.8. Function Component Scan

Spring Cloud Function will scan for implementations of `Function`, `Consumer` and `Supplier` in a package called `functions` if it exists. Using this feature you can write functions that have no dependencies on Spring - not even the `@Component` annotation is needed. If you want to use a different package, you can set `spring.cloud.function.scan.packages`. You can also use `spring.cloud.function.scan.enabled=false` to switch off the scan completely.

Chapter 62. Standalone Web Applications

Functions could be automatically exported as HTTP endpoints.

The `spring-cloud-function-web` module has autoconfiguration that activates when it is included in a Spring Boot web application (with MVC support). There is also a `spring-cloud-starter-function-web` to collect all the optional dependencies in case you just want a simple getting started experience.

With the web configurations activated your app will have an MVC endpoint (on "/" by default, but configurable with `spring.cloud.function.web.path`) that can be used to access the functions in the application context where function name becomes part of the URL path. The supported content types are plain text and JSON.

Method	Path	Request	Response	Status
GET	/{supplier}	-	Items from the named supplier	200 OK
POST	/{consumer}	JSON object or text	Mirrors input and pushes request body into consumer	202 Accepted
POST	/{consumer}	JSON array or text with new lines	Mirrors input and pushes body into consumer one by one	202 Accepted
POST	/{function}	JSON object or text	The result of applying the named function	200 OK
POST	/{function}	JSON array or text with new lines	The result of applying the named function	200 OK
GET	/{function}/{item}	-	Convert the item into an object and return the result of applying the function	200 OK

As the table above shows the behaviour of the endpoint depends on the method and also the type of incoming request data. When the incoming data is single valued, and the target function is declared as obviously single valued (i.e. not returning a collection or `Flux`), then the response will also contain a single value. For multi-valued responses the client can ask for a server-sent event stream by sending `Accept: text/event-stream".

Functions and consumers that are declared with input and output in `Message<?>` will see the request headers on the input messages, and the output message headers will be converted to HTTP headers.

When POSTing text the response format might be different with Spring Boot 2.0 and older versions,

depending on the content negotiation (provide content type and accept headers for the best results).

See Testing Functional Applications to see the details and example on how to test such application.

62.1. Function Mapping rules

If there is only a single function (consumer etc.) in the catalog, the name in the path is optional. In other words, providing you only have `uppercase` function in catalog `curl -H "Content-Type: text/plain" localhost:8080/uppercase -d hello` and `curl -H "Content-Type: text/plain" localhost:8080/ -d hello` calls are identical.

Composite functions can be addressed using pipes or commas to separate function names (pipes are legal in URL paths, but a bit awkward to type on the command line). For example, `curl -H "Content-Type: text/plain" localhost:8080/uppercase,reverse -d hello`.

For cases where there is more then a single function in catalog, each function will be exported and mapped with function name being part of the path (e.g., `localhost:8080/uppercase`). In this scenario you can still map specific function or function composition to the root path by providing `spring.cloud.function.definition` property

For example,

```
--spring.cloud.function.definition=foo|bar
```

The above property will compose 'foo' and 'bar' function and map the composed function to the "/" path.

62.2. Function Filtering rules

In situations where there are more then one function in catalog there may be a need to only export certain functions or function compositions. In that case you can use the same `spring.cloud.function.definition` property listing functions you intend to export delimited by `;`. Note that in this case nothing will be mapped to the root path and functions that are not listed (including compositions) are not going to be exported

For example,

```
--spring.cloud.function.definition=foo;bar
```

This will only export function `foo` and function `bar` regardless how many functions are available in catalog (e.g., `localhost:8080/foo`).

```
--spring.cloud.function.definition=foo|bar;baz
```

This will only export function composition `foo|bar` and function `baz` regardless how many functions

are available in catalog (e.g., `localhost:8080/foo,bar`).

Chapter 63. Standalone Streaming Applications

To send or receive messages from a broker (such as RabbitMQ or Kafka) you can leverage `spring-cloud-stream` project and it's integration with Spring Cloud Function. Please refer to Spring Cloud Function section of the Spring Cloud Stream reference manual for more details and examples.

Chapter 64. Deploying a Packaged Function

Spring Cloud Function provides a "deployer" library that allows you to launch a jar file (or exploded archive, or set of jar files) with an isolated class loader and expose the functions defined in it. This is quite a powerful tool that would allow you to, for instance, adapt a function to a range of different input-output adapters without changing the target jar file. Serverless platforms often have this kind of feature built in, so you could see it as a building block for a function invoker in such a platform (indeed the Riff Java function invoker uses this library).

The standard entry point is to add `spring-cloud-function-deployer` to the classpath, the deployer kicks in and looks for some configuration to tell it where to find the function jar.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-function-deployer</artifactId>
    <version>${spring.cloud.function.version}</version>
</dependency>
```

At a minimum the user has to provide a `spring.cloud.function.location` which is a URL or resource location for the archive containing the functions. It can optionally use a `maven:` prefix to locate the artifact via a dependency lookup (see `FunctionProperties` for complete details). A Spring Boot application is bootstrapped from the jar file, using the `MANIFEST.MF` to locate a start class, so that a standard Spring Boot fat jar works well, for example. If the target jar can be launched successfully then the result is a function registered in the main application's `FunctionCatalog`. The registered function can be applied by code in the main application, even though it was created in an isolated class loader (by deault).

Here is the example of deploying a JAR which contains an 'uppercase' function and invoking it .

```
@SpringBootApplication
public class DeployFunctionDemo {

    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(DeployFunctionDemo.class,
                "--spring.cloud.function.location=..../target/uppercase-0.0.1-
SNAPSHOT.jar",
                "--spring.cloud.function.definition=uppercase");

        FunctionCatalog catalog = context.getBean(FunctionCatalog.class);
        Function<String, String> function = catalog.lookup("uppercase");
        System.out.println(function.apply("hello"));
    }
}
```

And here is the example using Maven URI (taken from one of the tests in `FunctionDeployerTests`):

```
@SpringBootApplication
public class DeployFunctionDemo {

    public static void main(String[] args) {
        String[] args = new String[] {
                "--spring.cloud.function.location=maven://oz.demo:demo-
uppercase:0.0.1-SNAPSHOT",
                "--spring.cloud.function.function-class=oz.demo.uppercase.MyFunction"
};

        ApplicationContext context = SpringApplication.run(DeployerApplication.class,
args);
        FunctionCatalog catalog = context.getBean(FunctionCatalog.class);
        Function<String, String> function = catalog.lookup("myFunction");

        assertThat(function.apply("bob")).isEqualTo("BOB");
    }
}
```

Keep in mind that Maven resource such as local and remote repositories, user, password and more are resolved using default MavenProperties which effectively use local defaults and will work for majority of cases. However if you need to customize you can simply provide a bean of type `MavenProperties` where you can set additional properties (see example below).

```
@Bean
public MavenProperties mavenProperties() {
    MavenProperties properties = new MavenProperties();
    properties.setLocalRepository("target/it/");
    return properties;
}
```

64.1. Supported Packaging Scenarios

Currently Spring Cloud Function supports several packaging scenarios to give you the most flexibility when it comes to deploying functions.

64.1.1. Simple JAR

This packaging option implies no dependency on anything related to Spring. For example; Consider that such JAR contains the following class:

```
package function.example;
. . .
public class UpperCaseFunction implements Function<String, String> {
    @Override
    public String apply(String value) {
        return value.toUpperCase();
    }
}
```

All you need to do is specify `location` and `function-class` properties when deploying such package:

```
--spring.cloud.function.location=target/it/simplestjar/target/simplestjar
-1.0.0.RELEASE.jar
--spring.cloud.function.function-class=function.example.UpperCaseFunction
```

It's conceivable in some cases that you might want to package multiple functions together. For such scenarios you can use `spring.cloud.function.function-class` property to list several classes delimiting them by `;`.

For example,

```
--spring.cloud.function.function
-class=function.example.UpperCaseFunction;function.example.ReverseFunction
```

Here we are identifying two functions to deploy, which we can now access in function catalog by name (e.g., `catalog.lookup("reverseFunction");`).

For more details please reference the complete sample available here. You can also find a corresponding test in FunctionDeployerTests.

64.1.2. Spring Boot JAR

This packaging option implies there is a dependency on Spring Boot and that the JAR was generated as Spring Boot JAR. That said, given that the deployed JAR runs in the isolated class loader, there will not be any version conflict with the Spring Boot version used by the actual deployer. For example; Consider that such JAR contains the following class (which could have some additional Spring dependencies providing Spring/Spring Boot is on the classpath):

```
package function.example;
. . .
public class UpperCaseFunction implements Function<String, String> {
    @Override
    public String apply(String value) {
        return value.toUpperCase();
    }
}
```

As before all you need to do is specify `location` and `function-class` properties when deploying such package:

```
--spring.cloud.function.location=target/it/simplestjar/target/simplestjar
-1.0.0.RELEASE.jar
--spring.cloud.function.function-class=function.example.UpperCaseFunction
```

For more details please reference the complete sample available here. You can also find a corresponding test in FunctionDeployerTests.

64.1.3. Spring Boot Application

This packaging option implies your JAR is complete stand alone Spring Boot application with functions as managed Spring beans. As before there is an obvious assumption that there is a dependency on Spring Boot and that the JAR was generated as Spring Boot JAR. That said, given that the deployed JAR runs in the isolated class loader, there will not be any version conflict with the Spring Boot version used by the actual deployer. For example; Consider that such JAR contains the following class:

```
package function.example;
. . .
@SpringBootApplication
public class SimpleFunctionAppApplication {

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

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }
}
```

Given that we're effectively dealing with another Spring Application context and that functions are spring managed beans, in addition to the `location` property we also specify `definition` property instead of `function-class`.

```
--spring.cloud.function.location=target/it/bootapp/target/bootapp-1.0.0.RELEASE
-exec.jar
--spring.cloud.function.definition=uppercase
```

For more details please reference the complete sample available here. You can also find a corresponding test in FunctionDeployerTests.

This particular deployment option may or may not have Spring Cloud Function on it's classpath. From the deployer perspective this doesn't matter.

Chapter 65. Functional Bean Definitions

Spring Cloud Function supports a "functional" style of bean declarations for small apps where you need fast startup. The functional style of bean declaration was a feature of Spring Framework 5.0 with significant enhancements in 5.1.

65.1. Comparing Functional with Traditional Bean Definitions

Here's a vanilla Spring Cloud Function application from with the familiar `@Configuration` and `@Bean` declaration style:

```
@SpringBootApplication
public class DemoApplication {

  @Bean
  public Function<String, String> uppercase() {
    return value -> value.toUpperCase();
  }

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

}
```

Now for the functional beans: the user application code can be recast into "functional" form, like this:

```
@SpringBootConfiguration
public class DemoApplication implements
ApplicationContextInitializer<GenericApplicationContext> {

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

  public Function<String, String> uppercase() {
    return value -> value.toUpperCase();
  }

  @Override
  public void initialize(GenericApplicationContext context) {
    context.registerBean("demo", FunctionRegistration.class,
        () -> new FunctionRegistration<>(uppercase())
            .type(FunctionType.from(String.class).to(String.class)));
  }

}
```

The main differences are:

- The main class is an `ApplicationContextInitializer`.
- The `@Bean` methods have been converted to calls to `context.registerBean()`
- The `@SpringBootApplication` has been replaced with `@SpringBootConfiguration` to signify that we are not enabling Spring Boot autoconfiguration, and yet still marking the class as an "entry point".
- The `SpringApplication` from Spring Boot has been replaced with a `FunctionalSpringApplication` from Spring Cloud Function (it’s a subclass).

The business logic beans that you register in a Spring Cloud Function app are of type `FunctionRegistration`. This is a wrapper that contains both the function and information about the input and output types. In the `@Bean` form of the application that information can be derived reflectively, but in a functional bean registration some of it is lost unless we use a `FunctionRegistration`.

An alternative to using an `ApplicationContextInitializer` and `FunctionRegistration` is to make the application itself implement `Function` (or `Consumer` or `Supplier`). Example (equivalent to the above):

```
@SpringBootConfiguration
public class DemoApplication implements Function<String, String> {

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

  @Override
  public String apply(String value) {
    return value.toUpperCase();
  }

}
```

It would also work if you add a separate, standalone class of type `Function` and register it with the `SpringApplication` using an alternative form of the `run()` method. The main thing is that the generic type information is available at runtime through the class declaration.

Suppose you have

```
@Component
public class CustomFunction implements Function<Flux<Foo>, Flux<Bar>> {
    @Override
    public Flux<Bar> apply(Flux<Foo> flux) {
        return flux.map(foo -> new Bar("This is a Bar object from Foo value: " +
foo.getValue()));
    }

}
```

You register it as such:

```
@Override
public void initialize(GenericApplicationContext context) {
        context.registerBean("function", FunctionRegistration.class,
                () -> new FunctionRegistration<>(new
CustomFunction()).type(CustomFunction.class));
}
```

65.2. Limitations of Functional Bean Declaration

Most Spring Cloud Function apps have a relatively small scope compared to the whole of Spring Boot, so we are able to adapt it to these functional bean definitions easily. If you step outside that limited scope, you can extend your Spring Cloud Function app by switching back to `@Bean` style configuration, or by using a hybrid approach. If you want to take advantage of Spring Boot autoconfiguration for integrations with external datastores, for example, you will need to use

`@EnableAutoConfiguration`. Your functions can still be defined using the functional declarations if you want (i.e. the "hybrid" style), but in that case you will need to explicitly switch off the "full functional mode" using `spring.functional.enabled=false` so that Spring Boot can take back control.

Chapter 66. Testing Functional Applications

Spring Cloud Function also has some utilities for integration testing that will be very familiar to Spring Boot users.

Suppose this is your application:

```
@SpringBootApplication
public class SampleFunctionApplication {

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

    @Bean
    public Function<String, String> uppercase() {
        return v -> v.toUpperCase();
    }
}
```

Here is an integration test for the HTTP server wrapping this application:

```
@SpringBootTest(classes = SampleFunctionApplication.class,
            webEnvironment = WebEnvironment.RANDOM_PORT)
public class WebFunctionTests {

    @Autowired
    private TestRestTemplate rest;

    @Test
    public void test() throws Exception {
        ResponseEntity<String> result = this.rest.exchange(
            RequestEntity.post(new URI("/uppercase")).body("hello"), String.class);
        System.out.println(result.getBody());
    }
}
```

or when function bean definition style is used:

```
@FunctionalSpringBootTest
public class WebFunctionTests {

    @Autowired
    private TestRestTemplate rest;

    @Test
    public void test() throws Exception {
        ResponseEntity<String> result = this.rest.exchange(
            RequestEntity.post(new URI("/uppercase")).body("hello"), String.class);
        System.out.println(result.getBody());
    }
}
```

This test is almost identical to the one you would write for the `@Bean` version of the same app - the only difference is the `@FunctionalSpringBootTest` annotation, instead of the regular `@SpringBootTest`. All the other pieces, like the `@Autowired` `TestRestTemplate`, are standard Spring Boot features.

And to help with correct dependencies here is the excerpt from POM

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
. . . .
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-function-web</artifactId>
    <version>3.0.1.BUILD-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Or you could write a test for a non-HTTP app using just the `FunctionCatalog`. For example:

```
@RunWith(SpringRunner.class)
@FunctionalSpringBootTest
public class FunctionalTests {

    @Autowired
    private FunctionCatalog catalog;

    @Test
    public void words() throws Exception {
        Function<String, String> function = catalog.lookup(Function.class,
                "uppercase");
        assertThat(function.apply("hello")).isEqualTo("HELLO");
    }

}
```

Chapter 67. Dynamic Compilation

There is a sample app that uses the function compiler to create a function from a configuration property. The vanilla "function-sample" also has that feature. And there are some scripts that you can run to see the compilation happening at run time. To run these examples, change into the `scripts` directory:

```
cd scripts
```

Also, start a RabbitMQ server locally (e.g. execute `rabbitmq-server`).

Start the Function Registry Service:

```
./function-registry.sh
```

Register a Function:

```
./registerFunction.sh -n uppercase -f "f->f.map(s->s.toString().toUpperCase())"
```

Run a REST Microservice using that Function:

```
./web.sh -f uppercase -p 9000
curl -H "Content-Type: text/plain" -H "Accept: text/plain" localhost:9000/uppercase -d
foo
```

Register a Supplier:

```
./registerSupplier.sh -n words -f "()->Flux.just(\"foo\",\"bar\")"
```

Run a REST Microservice using that Supplier:

```
./web.sh -s words -p 9001
curl -H "Accept: application/json" localhost:9001/words
```

Register a Consumer:

```
./registerConsumer.sh -n print -t String -f "System.out::println"
```

Run a REST Microservice using that Consumer:

```
./web.sh -c print -p 9002
curl -X POST -H "Content-Type: text/plain" -d foo localhost:9002/print
```

Run Stream Processing Microservices:

First register a streaming words supplier:

```
./registerSupplier.sh -n wordstream -f "()-
>Flux.interval(Duration.ofMillis(1000)).map(i->\"message-\"+i)"
```

Then start the source (supplier), processor (function), and sink (consumer) apps (in reverse order):

```
./stream.sh -p 9103 -i uppercaseWords -c print
./stream.sh -p 9102 -i words -f uppercase -o uppercaseWords
./stream.sh -p 9101 -s wordstream -o words
```

The output will appear in the console of the sink app (one message per second, converted to uppercase):

```
MESSAGE-0
MESSAGE-1
MESSAGE-2
MESSAGE-3
MESSAGE-4
MESSAGE-5
MESSAGE-6
MESSAGE-7
MESSAGE-8
MESSAGE-9
...
```

Chapter 68. Serverless Platform Adapters

As well as being able to run as a standalone process, a Spring Cloud Function application can be adapted to run one of the existing serverless platforms. In the project there are adapters for AWS Lambda, Azure, and Apache OpenWhisk. The Oracle Fn platform has its own Spring Cloud Function adapter. And Riff supports Java functions and its Java Function Invoker acts natively is an adapter for Spring Cloud Function jars.

68.1. AWS Lambda

The AWS adapter takes a Spring Cloud Function app and converts it to a form that can run in AWS Lambda.

The details of how to get stared with AWS Lambda is out of scope of this document, so the expectation is that user has some familiarity with AWS and AWS Lambda and wants to learn what additional value spring provides.

68.1.1. Getting Started

One of the goals of Spring Cloud Function framework is to provide necessary infrastructure elements to enable a *simple function application* to interact in a certain way in a particular environment. A simple function application (in context or Spring) is an application that contains beans of type Supplier, Function or Consumer. So, with AWS it means that a simple function bean should somehow be recognised and executed in AWS Lambda environment.

Let’s look at the example:

```
@SpringBootApplication
public class FunctionConfiguration {

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

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }
}
```

It shows a complete Spring Boot application with a function bean defined in it. What’s interesting is that on the surface this is just another boot app, but in the context of AWS Adapter it is also a perfectly valid AWS Lambda application. No other code or configuration is required. All you need to do is package it and deploy it, so let’s look how we can do that.

To make things simpler we’ve provided a sample project ready to be built and deployed and you can access it here.

You simply execute `./mvnw clean package` to generate JAR file. All the necessary maven plugins have already been setup to generate appropriate AWS deployable JAR file. (You can read more details about JAR layout in Notes on JAR Layout).

Then you have to upload the JAR file (via AWS dashboard or AWS CLI) to AWS.

When ask about *handler* you specify `org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest` which is a generic request handler.

[AWS deploy] | *https://raw.githubusercontent.com/spring-cloud/spring-*

cloud/1.0.x/docs/src/main/asciidoc/images/AWS-deploy.png

That is all. Save and execute the function with some sample data which for this function is expected to be a String which function will uppercase and return back.

While `org.springframework.cloud.function.adapter.aws.FunctionInvoker` is a general purpose AWS's `RequestHandler` implementation aimed at completely isolating you from the specifics of AWS Lambda API, for some cases you may want to specify which specific AWS's `RequestHandler` you want to use. The next section will explain you how you can accomplish just that.

68.1.2. AWS Request Handlers

The adapter has a couple of generic request handlers that you can use. The most generic is (and the one we used in the Getting Started section) is `org.springframework.cloud.function.adapter.aws.FunctionInvoker` which is the implementation of AWS's `RequestStreamHandler`. User doesn't need to do anything other then specify it as 'handler' on AWS dashborad when deploying function. It will handle most of the case including Kinesis, streaming etc. .

If your app has more than one `@Bean` of type `Function` etc. then you can choose the one to use by configuring `spring.cloud.function.definition` property or environment variable. The functions are extracted from the Spring Cloud `FunctionCatalog`. In the event you don't specify `spring.cloud.function.definition` the framework will attempt to find a default following the search order where it searches first for `Function` then `Consumer` and finally `Supplier`).

68.1.3. Notes on JAR Layout

You don't need the Spring Cloud Function Web or Stream adapter at runtime in Lambda, so you might need to exclude those before you create the JAR you send to AWS. A Lambda application has to be shaded, but a Spring Boot standalone application does not, so you can run the same app using 2 separate jars (as per the sample). The sample app creates 2 jar files, one with an `aws` classifier for deploying in Lambda, and one executable (thin) jar that includes `spring-cloud-function-web` at runtime. Spring Cloud Function will try and locate a "main class" for you from the JAR file manifest, using the `Start-Class` attribute (which will be added for you by the Spring Boot tooling if you use the starter parent). If there is no `Start-Class` in your manifest you can use an environment variable or system property `MAIN_CLASS` when you deploy the function to AWS.

If you are not using the functional bean definitions but relying on Spring Boot's auto-configuration, then additional transformers must be configured as part of the maven-shade-plugin execution.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </dependency>
    </dependencies>
    <configuration>
        <createDependencyReducedPom>false</createDependencyReducedPom>
        <shadedArtifactAttached>true</shadedArtifactAttached>
        <shadedClassifierName>aws</shadedClassifierName>
        <transformers>
            <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
                <resource>META-INF/spring.handlers</resource>
            </transformer>
            <transformer
implementation="org.springframework.boot.maven.PropertiesMergingResourceTransformer">
                <resource>META-INF/spring.factories</resource>
            </transformer>
            <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
                <resource>META-INF/spring.schemas</resource>
            </transformer>
        </transformers>
    </configuration>
</plugin>
```

68.1.4. Build file setup

In order to run Spring Cloud Function applications on AWS Lambda, you can leverage Maven or Gradle plugins offered by the cloud platform provider.

Maven

In order to use the adapter plugin for Maven, add the plugin dependency to your `pom.xml` file:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-function-adapter-aws</artifactId>
    </dependency>
</dependencies>
```

As pointed out in the Notes on JAR Layout, you will need a shaded jar in order to upload it to AWS Lambda. You can use the Maven Shade Plugin for that. The example of the setup can be found

above.

You can use theSpring Boot Maven Plugin to generate the thin jar.

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot.experimental</groupId>
            <artifactId>spring-boot-thin-layout</artifactId>
            <version>${wrapper.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

You can find the entire sample `pom.xml` file for deploying Spring Cloud Function applications to AWS Lambda with Maven here.

Gradle

In order to use the adapter plugin for Gradle, add the dependency to your `build.gradle` file:

```
dependencies {
    compile("org.springframework.cloud:spring-cloud-function-adapter-aws:${version}")
}
```

As pointed out in Notes on JAR Layout, you will need a shaded jar in order to upload it to AWS Lambda. You can use the Gradle Shadow Plugin for that:

```
buildscript {
    dependencies {
        classpath "com.github.jengelman.gradle.plugins:shadow:${shadowPluginVersion}"
    }
}
apply plugin: 'com.github.johnrengelman.shadow'

assemble.dependsOn = [shadowJar]

import com.github.jengelman.gradle.plugins.shadow.transformers.*

shadowJar {
    classifier = 'aws'
    dependencies {
        exclude(
            dependency("org.springframework.cloud:spring-cloud-function-
web:${springCloudFunctionVersion}"))
    }
    // Required for Spring
    mergeServiceFiles()
    append 'META-INF/spring.handlers'
    append 'META-INF/spring.schemas'
    append 'META-INF/spring.tooling'
    transform(PropertiesFileTransformer) {
        paths = ['META-INF/spring.factories']
        mergeStrategy = "append"
    }
}
```

You can use the Spring Boot Gradle Plugin and Spring Boot Thin Gradle Plugin to generate the thin jar.

```
buildscript {
    dependencies {
        classpath("org.springframework.boot.experimental:spring-boot-thin-gradle-
plugin:${wrapperVersion}")
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}
apply plugin: 'org.springframework.boot'
apply plugin: 'org.springframework.boot.experimental.thin-launcher'
assemble.dependsOn = [thinJar]
```

You can find the entire sample `build.gradle` file for deploying Spring Cloud Function applications to AWS Lambda with Gradle here.

68.1.5. Upload

Build the sample under `spring-cloud-function-samples/function-sample-aws` and upload the `-aws` jar file to Lambda. The handler can be `example.Handler` or `org.springframework.cloud.function.adapter.aws.SpringBootStreamHandler` (FQN of the class, *not* a method reference, although Lambda does accept method references).

```
./mvnw -U clean package
```

Using the AWS command line tools it looks like this:

```
aws lambda create-function --function-name Uppercase --role
arn:aws:iam::[USERID]:role/service-role/[ROLE] --zip-file fileb://function-sample-
aws/target/function-sample-aws-2.0.0.BUILD-SNAPSHOT-aws.jar --handler
org.springframework.cloud.function.adapter.aws.SpringBootStreamHandler --description
"Spring Cloud Function Adapter Example" --runtime java8 --region us-east-1 --timeout
30 --memory-size 1024 --publish
```

The input type for the function in the AWS sample is a Foo with a single property called "value". So you would need this to test it:

```
{
  "value": "test"
}
```

The AWS sample app is written in the "functional" style (as an `ApplicationContextInitializer`). This is much faster on startup in Lambda than the traditional `@Bean` style, so if you don't need `@Beans` (or `@EnableAutoConfiguration`) it's a good choice. Warm starts are not affected.

68.1.6. Type Conversion

Spring Cloud Function will attempt to transparently handle type conversion between the raw input stream and types declared by your function.

For example, if your function signature is as such `Function<Foo, Bar>` we will attempt to convert incoming stream event to an instance of `Foo`.

In the event type is not known or can not be determined (e.g., `Function<?, ?>`) we will attempt to convert an incoming stream event to a generic `Map`.

Raw Input

There are times when you may want to have access to a raw input. In this case all you need is to declare your function signature to accept `InputStream`. For example, `Function<InputStream, ?>`. In this case we will not attempt any conversion and will pass the raw input directly to a function.

68.2. Microsoft Azure

The Azure adapter bootstraps a Spring Cloud Function context and channels function calls from the Azure framework into the user functions, using Spring Boot configuration where necessary. Azure Functions has quite a unique, but invasive programming model, involving annotations in user code that are specific to the platform. The easiest way to use it with Spring Cloud is to extend a base class and write a method in it with the `@FunctionName` annotation which delegates to a base class method.

This project provides an adapter layer for a Spring Cloud Function application onto Azure. You can write an app with a single `@Bean` of type `Function` and it will be deployable in Azure if you get the JAR file laid out right.

There is an `AzureSpringBootRequestHandler` which you must extend, and provide the input and output types as annotated method parameters (enabling Azure to inspect the class and create JSON bindings). The base class has two useful methods (`handleRequest` and `handleOutput`) to which you can delegate the actual function call, so mostly the function will only ever have one line.

Example:

```
public class FooHandler extends AzureSpringBootRequestHandler<Foo, Bar> {
    @FunctionName("uppercase")
    public Bar execute(@HttpTrigger(name = "req", methods = {HttpMethod.GET,
            HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
HttpRequestMessage<Optional<Foo>> request,
        ExecutionContext context) {
        return handleRequest(request.getBody().get(), context);
    }
}
```

This Azure handler will delegate to a `Function<Foo,Bar>` bean (or a `Function<Publisher<Foo>,Publisher<Bar>>`). Some Azure triggers (e.g. `@CosmosDBTrigger`) result in a input type of `List` and in that case you can bind to `List` in the Azure handler, or `String` (the raw JSON). The `List` input delegates to a `Function` with input type `Map<String,Object>`, or `Publisher` or `List` of the same type. The output of the `Function` can be a `List` (one-for-one) or a single value (aggregation), and the output binding in the Azure declaration should match.

If your app has more than one `@Bean` of type `Function` etc. then you can choose the one to use by configuring `function.name`. Or if you make the `@FunctionName` in the Azure handler method match the function name it should work that way (also for function apps with multiple functions). The functions are extracted from the Spring Cloud `FunctionCatalog` so the default function names are the same as the bean names.

68.2.1. Accessing Azure ExecutionContext

Some time there is a need to access the target execution context provided by Azure runtime in the form of `com.microsoft.azure.functions.ExecutionContext`. For example one of such needs is logging, so it can appear in the Azure console.

For that purpose Spring Cloud Function will register `ExecutionContext` as bean in the Application context, so it could be injected into your function. For example

```
@Bean
public Function<Foo, Bar> uppercase(ExecutionContext targetContext) {
    return foo -> {
        targetContext.getLogger().info("Invoking 'uppercase' on " + foo.getValue());
        return new Bar(foo.getValue().toUpperCase());
    };
}
```

Normally type-based injection should suffice, however if need to you can also utilise the bean name under which it is registered which is `targetExecutionContext`.

68.2.2. Notes on JAR Layout

You don't need the Spring Cloud Function Web at runtime in Azure, so you can exclude this before you create the JAR you deploy to Azure, but it won't be used if you include it, so it doesn't hurt to leave it in. A function application on Azure is an archive generated by the Maven plugin. The function lives in the JAR file generated by this project. The sample creates it as an executable jar, using the thin layout, so that Azure can find the handler classes. If you prefer you can just use a regular flat JAR file. The dependencies should **not** be included.

68.2.3. Build file setup

In order to run Spring Cloud Function applications on Microsoft Azure, you can leverage the Maven plugin offered by the cloud platform provider.

In order to use the adapter plugin for Maven, add the plugin dependency to your `pom.xml` file:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-function-adapter-azure</artifactId>
    </dependency>
</dependencies>
```

Then, configure the plugin. You will need to provide Azure-specific configuration for your application, specifying the `resourceGroup`, `appName` and other optional properties, and add the `package` goal execution so that the `function.json` file required by Azure is generated for you. Full plugin documentation can be found in the plugin repository.

```
<plugin>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure-functions-maven-plugin</artifactId>
    <configuration>
        <resourceGroup>${functionResourceGroup}</resourceGroup>
        <appName>${functionAppName}</appName>
    </configuration>
    <executions>
        <execution>
            <id>package-functions</id>
            <goals>
                <goal>package</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

You will also have to ensure that the files to be scanned by the plugin can be found in the Azure functions staging directory (see the plugin repository for more details on the staging directory and it's default location).

You can find the entire sample `pom.xml` file for deploying Spring Cloud Function applications to Microsoft Azure with Maven here.

As of yet, only Maven plugin is available. Gradle plugin has not been created by the cloud platform provider.

68.2.4. Build

```
./mvnw -U clean package
```

68.2.5. Running the sample

You can run the sample locally, just like the other Spring Cloud Function samples:

---

and `curl -H "Content-Type: text/plain" localhost:8080/api/uppercase -d '{"value": "hello foobar"}'`.

You will need the `az` CLI app (see docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-java-maven for more detail). To deploy the function on Azure runtime:

```
$ az login
$ mvn azure-functions:deploy
```

On another terminal try this: `curl <azure-function-url-from-the-log>/api/uppercase -d '{"value": "hello foobar!"}'`. Please ensure that you use the right URL for the function above. Alternatively you can test the function in the Azure Dashboard UI (click on the function name, go to the right hand side and click "Test" and to the bottom right, "Run").

The input type for the function in the Azure sample is a Foo with a single property called "value". So you need this to test it with something like below:

```
{
  "value": "foobar"
}
```

The Azure sample app is written in the "non-functional" style (using `@Bean`). The functional style (with just `Function` or `ApplicationContextInitializer`) is much faster on startup in Azure than the traditional `@Bean` style, so if you don't need `@Beans` (or `@EnableAutoConfiguration`) it's a good choice. Warm starts are not affected. :branch: master

68.3. Google Cloud Functions (Alpha)

The Google Cloud Functions adapter enables Spring Cloud Function apps to run on the Google Cloud Functions serverless platform. You can either run the function locally using the open source Google Functions Framework for Java or on GCP.

68.3.1. Project Dependencies

Start by adding the `spring-cloud-function-adapter-gcp` dependency to your project.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-function-adapter-gcp</artifactId>
    </dependency>

    ...
</dependencies>
```

In addition, add the `spring-boot-maven-plugin` which will build the JAR of the function to deploy.

Notice that we also reference `spring-cloud-function-adapter-gcp` as a dependency of the `spring-boot-maven-plugin`. This is necessary because it modifies the plugin to package your function in the correct JAR format for deployment on Google Cloud Functions.

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <outputDirectory>target/deploy</outputDirectory>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-function-adapter-gcp</artifactId>
        </dependency>
    </dependencies>
</plugin>
```

Finally, add the Maven plugin provided as part of the Google Functions Framework for Java. This allows you to test your functions locally via `mvn function:run`.

The function target should always be set to `org.springframework.cloud.function.adapter.gcp.GcfJarLauncher`; this is an adapter class which acts as the entry point to your Spring Cloud Function from the Google Cloud Functions platform.

```
<plugin>
    <groupId>com.google.cloud.functions</groupId>
    <artifactId>function-maven-plugin</artifactId>
    <version>0.9.1</version>
    <configuration>

<functionTarget>org.springframework.cloud.function.adapter.gcp.GcfJarLauncher</functio
nTarget>
        <port>8080</port>
    </configuration>
</plugin>
```

A full example of a working `pom.xml` can be found in the Spring Cloud Functions GCP sample.

68.3.2. HTTP Functions

Google Cloud Functions supports deploying HTTP Functions, which are functions that are invoked by HTTP request. The sections below describe instructions for deploying a Spring Cloud Function as an HTTP Function.

Getting Started

Let’s start with a simple Spring Cloud Function example:

```
@SpringBootApplication
public class CloudFunctionMain {

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

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }
}
```

Specify your configuration main class in `resources/META-INF/MANIFEST.MF`.

```
Main-Class: com.example.CloudFunctionMain
```

Then run the function locally. This is provided by the Google Cloud Functions `function-maven-plugin` described in the project dependencies section.

```
mvn function:run
```

Invoke the HTTP function:

```
curl http://localhost:8080/ -d "hello"
```

Deploy to GCP

As of March 2020, Google Cloud Functions for Java is in Alpha. You can get on the whitelist to try it out.

Start by packaging your application.

```
mvn package
```

If you added the custom `spring-boot-maven-plugin` plugin defined above, you should see the resulting JAR in `target/deploy` directory. This JAR is correctly formatted for deployment to Google Cloud Functions.

Next, make sure that you have the Cloud SDK CLI installed.

From the project base directory run the following command to deploy.

```
gcloud alpha functions deploy function-sample-gcp-http \
--entry-point org.springframework.cloud.function.adapter.gcp.GcfJarLauncher \
--runtime java11 \
--trigger-http \
--source target/deploy \
--memory 512MB
```

Invoke the HTTP function:

```
curl https://REGION-PROJECT_ID.cloudfunctions.net/function-sample-gcp-http -d "hello"
```

68.3.3. Background Functions

Google Cloud Functions also supports deploying Background Functions which are invoked indirectly in response to an event, such as a message on a Cloud Pub/Sub topic, a change in a Cloud Storage bucket, or a Firebase event.

The `spring-cloud-function-adapter-gcp` allows for functions to be deployed as background functions as well.

The sections below describe the process for writing a Cloud Pub/Sub topic background function. However, there are a number of different event types that can trigger a background function to execute which are not discussed here; these are described in the Background Function triggers documentation.

Getting Started

Let's start with a simple Spring Cloud Function which will run as a GCF background function:

```
@SpringBootApplication
public class BackgroundFunctionMain {

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

    @Bean
    public Consumer<PubSubMessage> pubSubFunction() {
        return message -> System.out.println("The Pub/Sub message data: " +
message.getData());
    }
}
```

In addition, create `PubSubMessage` class in the project with the below definition. This class represents the Pub/Sub event structure which gets passed to your function on a Pub/Sub topic event.

```
public class PubSubMessage {

    private String data;

    private Map<String, String> attributes;

    private String messageId;

    private String publishTime;

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }

    public Map<String, String> getAttributes() {
        return attributes;
    }

    public void setAttributes(Map<String, String> attributes) {
        this.attributes = attributes;
    }

    public String getMessageId() {
        return messageId;
    }

    public void setMessageId(String messageId) {
        this.messageId = messageId;
    }

    public String getPublishTime() {
        return publishTime;
    }

    public void setPublishTime(String publishTime) {
        this.publishTime = publishTime;
    }

}
```

Specify your configuration main class in `resources/META-INF/MANIFEST.MF`.

```
Main-Class: com.example.BackgroundFunctionMain
```

Then run the function locally. This is provided by the Google Cloud Functions `function-maven-plugin` described in the project dependencies section.

```
mvn function:run
```

Invoke the HTTP function:

```
curl localhost:8080 -H "Content-Type: application/json" -d '{"data":"hello"}'
```

Verify that the function was invoked by viewing the logs.

Deploy to GCP

In order to deploy your background function to GCP, first package your application.

```
mvn package
```

If you added the custom `spring-boot-maven-plugin` plugin defined above, you should see the resulting JAR in `target/deploy` directory. This JAR is correctly formatted for deployment to Google Cloud Functions.

Next, make sure that you have the Cloud SDK CLI installed.

From the project base directory run the following command to deploy.

```
gcloud alpha functions deploy function-sample-gcp-background \
--entry-point org.springframework.cloud.function.adapter.gcp.GcfJarLauncher \
--runtime java11 \
--trigger-topic my-functions-topic \
--source target/deploy \
--memory 512MB
```

Google Cloud Function will now invoke the function every time a message is published to the topic specified by `--trigger-topic`.

For a walkthrough on testing and verifying your background function, see the instructions for running the GCF Background Function sample.

68.3.4. Sample Functions

The project provides the following sample functions as reference:

- The function-sample-gcp-http is an HTTP Function which you can test locally and try deploying.
- The function-sample-gcp-background shows an example of a background function that is triggered by a message being published to a specified Pub/Sub topic.

Spring Cloud Gateway

Hoxton.SR8

This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2 and Project Reactor. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

Chapter 69. How to Include Spring Cloud Gateway

To include Spring Cloud Gateway in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-gateway`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

If you include the starter, but you do not want the gateway to be enabled, set `spring.cloud.gateway.enabled=false`.

Spring Cloud Gateway is built on Spring Boot 2.x, Spring WebFlux, and Project Reactor. As a consequence, many of the familiar synchronous libraries (Spring Data and Spring Security, for example) and patterns you know may not apply when you use Spring Cloud Gateway. If you are unfamiliar with these projects, we suggest you begin by reading their documentation to familiarize yourself with some of the new concepts before working with Spring Cloud Gateway.

Spring Cloud Gateway requires the Netty runtime provided by Spring Boot and Spring Webflux. It does not work in a traditional Servlet Container or when built as a WAR.

Chapter 70. Glossary

- **Route**: The basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates, and a collection of filters. A route is matched if the aggregate predicate is true.
- **Predicate**: This is a Java 8 Function Predicate. The input type is a Spring Framework `ServerWebExchange`. This lets you match on anything from the HTTP request, such as headers or parameters.
- **Filter**: These are instances of Spring Framework `GatewayFilter` that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.

Chapter 71. How It Works

The following diagram provides a high-level overview of how Spring Cloud Gateway works:

[Spring Cloud Gateway Diagram] | *spring_cloud_gateway_diagram.png*

Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a route, it is sent to the Gateway Web Handler. This handler runs the request through a filter chain that is specific to the request. The reason the filters are divided by the dotted line is that filters can run logic both before and after the proxy request is sent. All "pre" filter logic is executed. Then the proxy request is made. After the proxy request is made, the "post" filter logic is run.

URIs defined in routes without a port get default port values of 80 and 443 for the HTTP and HTTPS URIs, respectively.

Chapter 72. Configuring Route Predicate Factories and Gateway Filter Factories

There are two ways to configure predicates and filters: shortcuts and fully expanded arguments. Most examples below use the shortcut way.

The name and argument names will be listed as `code` in the first sentance or two of the each section. The arguments are typically listed in the order that would be needed for the shortcut configuration.

72.1. Shortcut Configuration

Shortcut configuration is recognized by the filter name, followed by an equals sign (`=`), followed by argument values separated by commas (`,`).

application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: https://example.org
        predicates:
        - Cookie=mycookie,mycookievalue
```

The previous sample defines the `Cookie` Route Predicate Factory with two arguments, the cookie name, `mycookie` and the value to match `mycookievalue`.

72.2. Fully Expanded Arguments

Fully expanded arguments appear more like standard yaml configuration with name/value pairs. Typically, there will be a `name` key and an `args` key. The `args` key is a map of key value pairs to configure the predicate or filter.

application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: https://example.org
        predicates:
        - name: Cookie
          args:
            name: mycookie
            regexp: mycookievalue
```

This is the full configuration of the shortcut configuration of the `Cookie` predicate shown above.

Chapter 73. Route Predicate Factories

Spring Cloud Gateway matches routes as part of the Spring WebFlux `HandlerMapping` infrastructure. Spring Cloud Gateway includes many built-in route predicate factories. All of these predicates match on different attributes of the HTTP request. You can combine multiple route predicate factories with logical `and` statements.

73.1. The After Route Predicate Factory

The `After` route predicate factory takes one parameter, a `datetime` (which is a java `ZonedDateTime`). This predicate matches requests that happen after the specified datetime. The following example configures an after route predicate:

Example 5. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: https://example.org
        predicates:
        - After=2017-01-20T17:42:47.789-07:00[America/Denver]
```

This route matches any request made after Jan 20, 2017 17:42 Mountain Time (Denver).

73.2. The Before Route Predicate Factory

The `Before` route predicate factory takes one parameter, a `datetime` (which is a java `ZonedDateTime`). This predicate matches requests that happen before the specified `datetime`. The following example configures a before route predicate:

Example 6. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: before_route
        uri: https://example.org
        predicates:
        - Before=2017-01-20T17:42:47.789-07:00[America/Denver]
```

This route matches any request made before Jan 20, 2017 17:42 Mountain Time (Denver).

73.3. The Between Route Predicate Factory

The `Between` route predicate factory takes two parameters, `datetime1` and `datetime2` which are java `ZonedDateTime` objects. This predicate matches requests that happen after `datetime1` and before `datetime2`. The `datetime2` parameter must be after `datetime1`. The following example configures a between route predicate:

Example 7. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: between_route
        uri: https://example.org
        predicates:
        - Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-
21T17:42:47.789-07:00[America/Denver]
```

This route matches any request made after Jan 20, 2017 17:42 Mountain Time (Denver) and before Jan 21, 2017 17:42 Mountain Time (Denver). This could be useful for maintenance windows.

73.4. The Cookie Route Predicate Factory

The `Cookie` route predicate factory takes two parameters, the cookie `name` and a `regexp` (which is a Java regular expression). This predicate matches cookies that have the given name and whose values match the regular expression. The following example configures a cookie route predicate factory:

Example 8. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: cookie_route
        uri: https://example.org
        predicates:
        - Cookie=chocolate, ch.p
```

This route matches requests that have a cookie named `chocolate` whose value matches the `ch.p` regular expression.

73.5. The Header Route Predicate Factory

The `Header` route predicate factory takes two parameters, the header `name` and a `regexp` (which is a Java regular expression). This predicate matches with a header that has the given name whose value matches the regular expression. The following example configures a header route predicate:

Example 9. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: header_route
        uri: https://example.org
        predicates:
        - Header=X-Request-Id, \d+
```

This route matches if the request has a header named `X-Request-Id` whose value matches the `\d+` regular expression (that is, it has a value of one or more digits).

73.6. The Host Route Predicate Factory

The `Host` route predicate factory takes one parameter: a list of host name `patterns`. The pattern is an Ant-style pattern with `.` as the separator. This predicates matches the `Host` header that matches the pattern. The following example configures a host route predicate:

Example 10. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: host_route
        uri: https://example.org
        predicates:
        - Host=**.somehost.org,**.anotherhost.org
```

URI template variables (such as `{sub}.myhost.org`) are supported as well.

This route matches if the request has a `Host` header with a value of `www.somehost.org` or `beta.somehost.org` or `www.anotherhost.org`.

This predicate extracts the URI template variables (such as `sub`, defined in the preceding example) as a map of names and values and places it in the `ServerWebExchange.getAttributes()` with a key defined in `ServerWebExchangeUtils.URI_TEMPLATE_VARIABLES_ATTRIBUTE`. Those values are then

available for use by `GatewayFilter` factories

73.7. The Method Route Predicate Factory

The `Method` Route Predicate Factory takes a `methods` argument which is one or more parameters: the HTTP methods to match. The following example configures a method route predicate:

Example 11. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: method_route
        uri: https://example.org
        predicates:
        - Method=GET,POST
```

This route matches if the request method was a `GET` or a `POST`.

73.8. The Path Route Predicate Factory

The `Path` Route Predicate Factory takes two parameters: a list of Spring `PathMatcher` `patterns` and an optional flag called `matchOptionalTrailingSeparator`. The following example configures a path route predicate:

Example 12. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: path_route
        uri: https://example.org
        predicates:
        - Path=/red/{segment},/blue/{segment}
```

This route matches if the request path was, for example: `/red/1` or `/red/blue` or `/blue/green`.

This predicate extracts the URI template variables (such as `segment`, defined in the preceding example) as a map of names and values and places it in the `ServerWebExchange.getAttributes()` with a key defined in `ServerWebExchangeUtils.URI_TEMPLATE_VARIABLES_ATTRIBUTE`. Those values are then available for use by `GatewayFilter` factories

A utility method (called `get`) is available to make access to these variables easier. The following

example shows how to use the `get` method:

```
Map<String, String> uriVariables =
ServerWebExchangeUtils.getPathPredicateVariables(exchange);

String segment = uriVariables.get("segment");
```

73.9. The Query Route Predicate Factory

The `Query` route predicate factory takes two parameters: a required `param` and an optional `regexp` (which is a Java regular expression). The following example configures a query route predicate:

Example 13. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: https://example.org
        predicates:
        - Query=green
```

The preceding route matches if the request contained a `green` query parameter.

application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: https://example.org
        predicates:
        - Query=red, gree.
```

The preceding route matches if the request contained a `red` query parameter whose value matched the `gree.` regexp, so `green` and `greet` would match.

73.10. The RemoteAddr Route Predicate Factory

The `RemoteAddr` route predicate factory takes a list (min size 1) of `sources`, which are CIDR-notation (IPv4 or IPv6) strings, such as `192.168.0.1/16` (where `192.168.0.1` is an IP address and `16` is a subnet mask). The following example configures a RemoteAddr route predicate:

Example 14. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: remoteaddr_route
        uri: https://example.org
        predicates:
        - RemoteAddr=192.168.1.1/24
```

This route matches if the remote address of the request was, for example, `192.168.1.10`.

73.11. The Weight Route Predicate Factory

The `Weight` route predicate factory takes two arguments: `group` and `weight` (an int). The weights are calculated per group. The following example configures a weight route predicate:

Example 15. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: weight_high
        uri: https://weighthigh.org
        predicates:
        - Weight=group1, 8
      - id: weight_low
        uri: https://weightlow.org
        predicates:
        - Weight=group1, 2
```

This route would forward ~80% of traffic to weighthigh.org and ~20% of traffic to weighlow.org

73.11.1. Modifying the Way Remote Addresses Are Resolved

By default, the RemoteAddr route predicate factory uses the remote address from the incoming request. This may not match the actual client IP address if Spring Cloud Gateway sits behind a proxy layer.

You can customize the way that the remote address is resolved by setting a custom `RemoteAddressResolver`. Spring Cloud Gateway comes with one non-default remote address resolver that is based off of the X-Forwarded-For header, `XForwardedRemoteAddressResolver`.

`XForwardedRemoteAddressResolver` has two static constructor methods, which take different

approaches to security:

- `XForwardedRemoteAddressResolver::trustAll` returns a `RemoteAddressResolver` that always takes the first IP address found in the `X-Forwarded-For` header. This approach is vulnerable to spoofing, as a malicious client could set an initial value for the `X-Forwarded-For`, which would be accepted by the resolver.
- `XForwardedRemoteAddressResolver::maxTrustedIndex` takes an index that correlates to the number of trusted infrastructure running in front of Spring Cloud Gateway. If Spring Cloud Gateway is, for example only accessible through HAProxy, then a value of 1 should be used. If two hops of trusted infrastructure are required before Spring Cloud Gateway is accessible, then a value of 2 should be used.

Consider the following header value:

```
X-Forwarded-For: 0.0.0.1, 0.0.0.2, 0.0.0.3
```

The following `maxTrustedIndex` values yield the following remote addresses:

`maxTrustedIndex`	**result**
[`Integer.MIN_VALUE`,0]	(invalid, `IllegalArgumentException` during initialization)
1	0.0.0.3
2	0.0.0.2
3	0.0.0.1
[4, `Integer.MAX_VALUE`]	0.0.0.1

The following example shows how to achieve the same configuration with Java:

Example 16. GatewayConfig.java

```
RemoteAddressResolver resolver = XForwardedRemoteAddressResolver
    .maxTrustedIndex(1);

...

.route("direct-route",
    r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24")
        .uri("https://downstream1")
.route("proxied-route",
    r -> r.remoteAddr(resolver, "10.10.1.1", "10.10.1.1/24")
        .uri("https://downstream2")
)
```

Chapter 74. GatewayFilter Factories

Route filters allow the modification of the incoming HTTP request or outgoing HTTP response in some manner. Route filters are scoped to a particular route. Spring Cloud Gateway includes many built-in GatewayFilter Factories.

For more detailed examples of how to use any of the following filters, take a look at the unit tests.

74.1. The AddRequestHeader GatewayFilter Factory

The AddRequestHeader GatewayFilter factory takes a name and value parameter. The following example configures an AddRequestHeader GatewayFilter:

Example 17. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: https://example.org
        filters:
        - AddRequestHeader=X-Request-red, blue
```

This listing adds X-Request-red:blue header to the downstream request's headers for all matching requests.

AddRequestHeader is aware of the URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an AddRequestHeader GatewayFilter that uses a variable:

Example 18. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: https://example.org
        predicates:
        - Path=/red/{segment}
        filters:
        - AddRequestHeader=X-Request-Red, Blue-{segment}
```

74.2. The `AddRequestParameter` `GatewayFilter` Factory

The `AddRequestParameter` `GatewayFilter` Factory takes a `name` and `value` parameter. The following example configures an `AddRequestParameter` `GatewayFilter`:

Example 19. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_parameter_route
        uri: https://example.org
        filters:
        - AddRequestParameter=red, blue
```

This will add `red=blue` to the downstream request's query string for all matching requests.

`AddRequestParameter` is aware of the URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `AddRequestParameter` `GatewayFilter` that uses a variable:

Example 20. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_parameter_route
        uri: https://example.org
        predicates:
        - Host: {segment}.myhost.org
        filters:
        - AddRequestParameter=foo, bar-{segment}
```

74.3. The `AddResponseHeader` `GatewayFilter` Factory

The `AddResponseHeader` `GatewayFilter` Factory takes a `name` and `value` parameter. The following example configures an `AddResponseHeader` `GatewayFilter`:

Example 21. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_response_header_route
        uri: https://example.org
        filters:
        - AddResponseHeader=X-Response-Red, Blue
```

This adds `X-Response-Foo:Bar` header to the downstream response's headers for all matching requests.

`AddResponseHeader` is aware of URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `AddResponseHeader GatewayFilter` that uses a variable:

Example 22. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_response_header_route
        uri: https://example.org
        predicates:
        - Host: {segment}.myhost.org
        filters:
        - AddResponseHeader=foo, bar-{segment}
```

74.4. The `DedupeResponseHeader GatewayFilter` Factory

The DedupeResponseHeader GatewayFilter factory takes a `name` parameter and an optional `strategy` parameter. `name` can contain a space-separated list of header names. The following example configures a `DedupeResponseHeader GatewayFilter`:

Example 23. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: dedupe_response_header_route
        uri: https://example.org
        filters:
        - DedupeResponseHeader=Access-Control-Allow-Credentials Access-Control-
Allow-Origin
```

This removes duplicate values of `Access-Control-Allow-Credentials` and `Access-Control-Allow-Origin` response headers in cases when both the gateway CORS logic and the downstream logic add them.

The `DedupeResponseHeader` filter also accepts an optional `strategy` parameter. The accepted values are `RETAIN_FIRST` (default), `RETAIN_LAST`, and `RETAIN_UNIQUE`.

74.5. The Hystrix `GatewayFilter` Factory

Netflix has put Hystrix in maintenance mode. We suggest you use the Spring Cloud CircuitBreaker Gateway Filter with Resilience4J, as support for Hystrix will be removed in a future release.

Hystrix is a library from Netflix that implements the circuit breaker pattern. The `Hystrix` `GatewayFilter` lets you introduce circuit breakers to your gateway routes, protecting your services from cascading failures and letting you provide fallback responses in the event of downstream failures.

To enable `Hystrix GatewayFilter` instances in your project, add a dependency on `spring-cloud-starter-netflix-hystrix` from Spring Cloud Netflix.

The `Hystrix GatewayFilter` factory requires a single `name` parameter, which is the name of the `HystrixCommand`. The following example configures a Hystrix `GatewayFilter`:

Example 24. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: hystrix_route
        uri: https://example.org
        filters:
        - Hystrix=myCommandName
```

This wraps the remaining filters in a `HystrixCommand` with a command name of `myCommandName`.

The Hystrix filter can also accept an optional `fallbackUri` parameter. Currently, only `forward:` schemed URIs are supported. If the fallback is called, the request is forwarded to the controller matched by the URI. The following example configures such a fallback:

Example 25. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: hystrix_route
        uri: lb://backing-service:8088
        predicates:
        - Path=/consumingserviceendpoint
        filters:
        - name: Hystrix
          args:
            name: fallbackcmd
            fallbackUri: forward:/incaseoffailureusethis
        - RewritePath=/consumingserviceendpoint, /backingserviceendpoint
```

This will forward to the `/incaseoffailureusethis` URI when the Hystrix fallback is called. Note that this example also demonstrates (optional) Spring Cloud Netflix Ribbon load-balancing (defined the `lb` prefix on the destination URI).

The primary scenario is to use the `fallbackUri` to an internal controller or handler within the gateway app. However, you can also reroute the request to a controller or handler in an external application, as follows:

Example 26. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: ingredients
        uri: lb://ingredients
        predicates:
        - Path=//ingredients/**
        filters:
        - name: Hystrix
          args:
            name: fetchIngredients
            fallbackUri: forward:/fallback
      - id: ingredients-fallback
        uri: http://localhost:9994
        predicates:
        - Path=/fallback
```

In this example, there is no `fallback` endpoint or handler in the gateway application. However, there is one in another application, registered under `localhost:9994`.

In case of the request being forwarded to the fallback, the Hystrix Gateway filter also provides the `Throwable` that has caused it. It is added to the `ServerWebExchange` as the `ServerWebExchangeUtils.HYSTRIX_EXECUTION_EXCEPTION_ATTR` attribute, which you can use when handling the fallback within the gateway application.

For the external controller/handler scenario, you can add headers with exception details. You can find more information on doing so in the FallbackHeaders GatewayFilter Factory section.

You can configured Hystrix settings (such as timeouts) with global defaults or on a route-by-route basis by using application properties, as explained on the Hystrix wiki.

To set a five-second timeout for the example route shown earlier, you could use the following configuration:

Example 27. application.yml

```
hystrix.command.fallbackcmd.execution.isolation.thread.timeoutInMilliseconds: 5000
```

74.6. Spring Cloud CircuitBreaker GatewayFilter Factory

The Spring Cloud CircuitBreaker GatewayFilter factory uses the Spring Cloud CircuitBreaker APIs to

wrap Gateway routes in a circuit breaker. Spring Cloud CircuitBreaker supports two libraries that can be used with Spring Cloud Gateway, Hystrix and Resilience4J. Since Netflix has placed Hystrix in maintenance-only mode, we suggest that you use Resilience4J.

To enable the Spring Cloud CircuitBreaker filter, you need to place either `spring-cloud-starter-circuitbreaker-reactor-resilience4j` or `spring-cloud-starter-netflix-hystrix` on the classpath. The following example configures a Spring Cloud CircuitBreaker `GatewayFilter`:

Example 28. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: circuitbreaker_route
        uri: https://example.org
        filters:
        - CircuitBreaker=myCircuitBreaker
```

To configure the circuit breaker, see the configuration for the underlying circuit breaker implementation you are using.

- Resilience4J Documentation
- Hystrix Documentation

The Spring Cloud CircuitBreaker filter can also accept an optional `fallbackUri` parameter. Currently, only `forward:` schemed URIs are supported. If the fallback is called, the request is forwarded to the controller matched by the URI. The following example configures such a fallback:

Example 29. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: circuitbreaker_route
        uri: lb://backing-service:8088
        predicates:
        - Path=/consumingServiceEndpoint
        filters:
        - name: CircuitBreaker
          args:
            name: myCircuitBreaker
            fallbackUri: forward:/inCaseOfFailureUseThis
        - RewritePath=/consumingServiceEndpoint, /backingServiceEndpoint
```

The following listing does the same thing in Java:

Example 30. Application.java

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("circuitbreaker_route", r -> r.path("/consumingServiceEndpoint")
            .filters(f -> f.circuitBreaker(c ->
c.name("myCircuitBreaker").fallbackUri("forward:/inCaseOfFailureUseThis"))
                .rewritePath("/consumingServiceEndpoint",
"/backingServiceEndpoint")).uri("lb://backing-service:8088")
        .build();
}
```

This example forwards to the `/inCaseofFailureUseThis` URI when the circuit breaker fallback is called. Note that this example also demonstrates the (optional) Spring Cloud Netflix Ribbon load-balancing (defined by the `lb` prefix on the destination URI).

The primary scenario is to use the `fallbackUri` to define an internal controller or handler within the gateway application. However, you can also reroute the request to a controller or handler in an external application, as follows:

Example 31. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: ingredients
        uri: lb://ingredients
        predicates:
        - Path=//ingredients/**
        filters:
        - name: CircuitBreaker
          args:
            name: fetchIngredients
            fallbackUri: forward:/fallback
      - id: ingredients-fallback
        uri: http://localhost:9994
        predicates:
        - Path=/fallback
```

In this example, there is no `fallback` endpoint or handler in the gateway application. However, there is one in another application, registered under `localhost:9994`.

In case of the request being forwarded to fallback, the Spring Cloud CircuitBreaker Gateway filter

also provides the `Throwable` that has caused it. It is added to the `ServerWebExchange` as the `ServerWebExchangeUtils.CIRCUITBREAKER_EXECUTION_EXCEPTION_ATTR` attribute that can be used when handling the fallback within the gateway application.

For the external controller/handler scenario, headers can be added with exception details. You can find more information on doing so in the FallbackHeaders GatewayFilter Factory section.

74.6.1. Tripping The Circuit Breaker On Status Codes

In some cases you might want to trip a circuit breaker based on the status code returned from the route it wraps. The circuit breaker config object takes a list of status codes that if returned will cause the the circuit breaker to be tripped. When setting the status codes you want to trip the circuit breaker you can either use a integer with the status code value or the String representation of the `HttpStatus` enumeration.

Example 32. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: circuitbreaker_route
        uri: lb://backing-service:8088
        predicates:
        - Path=/consumingServiceEndpoint
        filters:
        - name: CircuitBreaker
          args:
            name: myCircuitBreaker
            fallbackUri: forward:/inCaseOfFailureUseThis
            statusCodes:
              - 500
              - "NOT_FOUND"
```

Example 33. Application.java

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("circuitbreaker_route", r -> r.path("/consumingServiceEndpoint")
            .filters(f -> f.circuitBreaker(c ->
c.name("myCircuitBreaker").fallbackUri("forward:/inCaseOfFailureUseThis").addStatu
sCode("INTERNAL_SERVER_ERROR"))
                .rewritePath("/consumingServiceEndpoint",
"/backingServiceEndpoint")).uri("lb://backing-service:8088")
        .build();
}
```

74.7. The `FallbackHeaders` `GatewayFilter` Factory

The `FallbackHeaders` factory lets you add Hystrix or Spring Cloud CircuitBreaker execution exception details in the headers of a request forwarded to a `fallbackUri` in an external application, as in the following scenario:

Example 34. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: ingredients
        uri: lb://ingredients
        predicates:
        - Path=//ingredients/**
        filters:
        - name: CircuitBreaker
          args:
            name: fetchIngredients
            fallbackUri: forward:/fallback
      - id: ingredients-fallback
        uri: http://localhost:9994
        predicates:
        - Path=/fallback
        filters:
        - name: FallbackHeaders
          args:
            executionExceptionTypeHeaderName: Test-Header
```

In this example, after an execution exception occurs while running the circuit breaker, the request is forwarded to the `fallback` endpoint or handler in an application running on `localhost:9994`. The

headers with the exception type, message and (if available) root cause exception type and message are added to that request by the `FallbackHeaders` filter.

You can overwrite the names of the headers in the configuration by setting the values of the following arguments (shown with their default values):

- `executionExceptionTypeHeaderName` (`"Execution-Exception-Type"`)
- `executionExceptionMessageHeaderName` (`"Execution-Exception-Message"`)
- `rootCauseExceptionTypeHeaderName` (`"Root-Cause-Exception-Type"`)
- `rootCauseExceptionMessageHeaderName` (`"Root-Cause-Exception-Message"`)

For more information on circuit breakers and the gateway see the Hystrix GatewayFilter Factory section or Spring Cloud CircuitBreaker Factory section.

74.8. The `MapRequestHeader` `GatewayFilter` Factory

The `MapRequestHeader GatewayFilter` factory takes `fromHeader` and `toHeader` parameters. It creates a new named header (`toHeader`), and the value is extracted out of an existing named header (`fromHeader`) from the incoming http request. If the input header does not exist, the filter has no impact. If the new named header already exists, its values are augmented with the new values. The following example configures a `MapRequestHeader`:

Example 35. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: map_request_header_route
        uri: https://example.org
        filters:
        - MapRequestHeader=Blue, X-Request-Red
```

This adds `X-Request-Red:<values>` header to the downstream request with updated values from the incoming HTTP request's `Blue` header.

74.9. The `PrefixPath` `GatewayFilter` Factory

The `PrefixPath GatewayFilter` factory takes a single `prefix` parameter. The following example configures a `PrefixPath GatewayFilter`:

Example 36. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: https://example.org
        filters:
        - PrefixPath=/mypath
```

This will prefix `/mypath` to the path of all matching requests. So a request to `/hello` would be sent to `/mypath/hello`.

74.10. The `PreserveHostHeader` `GatewayFilter` Factory

The `PreserveHostHeader` `GatewayFilter` factory has no parameters. This filter sets a request attribute that the routing filter inspects to determine if the original host header should be sent, rather than the host header determined by the HTTP client. The following example configures a `PreserveHostHeader` `GatewayFilter`:

Example 37. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: preserve_host_route
        uri: https://example.org
        filters:
        - PreserveHostHeader
```

74.11. The `RequestRateLimiter` `GatewayFilter` Factory

The `RequestRateLimiter` `GatewayFilter` factory uses a `RateLimiter` implementation to determine if the current request is allowed to proceed. If it is not, a status of `HTTP 429 - Too Many Requests` (by default) is returned.

This filter takes an optional `keyResolver` parameter and parameters specific to the rate limiter (described later in this section).

`keyResolver` is a bean that implements the `KeyResolver` interface. In configuration, reference the bean by name using SpEL. `#{@myKeyResolver}` is a SpEL expression that references a bean named `myKeyResolver`. The following listing shows the `KeyResolver` interface:

Example 38. KeyResolver.java

```
public interface KeyResolver {
    Mono<String> resolve(ServerWebExchange exchange);
}
```

The `KeyResolver` interface lets pluggable strategies derive the key for limiting requests. In future milestone releases, there will be some `KeyResolver` implementations.

The default implementation of `KeyResolver` is the `PrincipalNameKeyResolver`, which retrieves the `Principal` from the `ServerWebExchange` and calls `Principal.getName()`.

By default, if the `KeyResolver` does not find a key, requests are denied. You can adjust this behavior by setting the `spring.cloud.gateway.filter.request-rate-limiter.deny-empty-key` (`true` or `false`) and `spring.cloud.gateway.filter.request-rate-limiter.empty-key-status-code` properties.

The `RequestRateLimiter` is not configurable with the "shortcut" notation. The following example below is *invalid*:

Example 39. application.properties

```
# INVALID SHORTCUT CONFIGURATION
spring.cloud.gateway.routes[0].filters[0]=RequestRateLimiter=2, 2,
#{@userkeyresolver}
```

74.11.1. The Redis `RateLimiter`

The Redis implementation is based off of work done at Stripe. It requires the use of the `spring-boot-starter-data-redis-reactive` Spring Boot starter.

The algorithm used is the Token Bucket Algorithm.

The `redis-rate-limiter.replenishRate` property is how many requests per second you want a user to be allowed to do, without any dropped requests. This is the rate at which the token bucket is filled.

The `redis-rate-limiter.burstCapacity` property is the maximum number of requests a user is allowed to do in a single second. This is the number of tokens the token bucket can hold. Setting this value to zero blocks all requests.

The `redis-rate-limiter.requestedTokens` property is how many tokens a request costs. This is the number of tokens taken from the bucket for each request and defaults to `1`.

A steady rate is accomplished by setting the same value in `replenishRate` and `burstCapacity`. Temporary bursts can be allowed by setting `burstCapacity` higher than `replenishRate`. In this case, the rate limiter needs to be allowed some time between bursts (according to `replenishRate`), as two

consecutive bursts will result in dropped requests (`HTTP 429 - Too Many Requests`). The following listing configures a `redis-rate-limiter`:

Rate limits bellow `1 request/s` are accomplished by setting `replenishRate` to the wanted number of requests, `requestedTokens` to the timespan in seconds and `burstCapacity` to the product of `replenishRate` and `requestedTokens`, e.g. setting `replenishRate=1`, `requestedTokens=60` and `burstCapacity=60` will result in a limit of `1 request/min`.

Example 40. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: requestratelimiter_route
        uri: https://example.org
        filters:
        - name: RequestRateLimiter
          args:
            redis-rate-limiter.replenishRate: 10
            redis-rate-limiter.burstCapacity: 20
            redis-rate-limiter.requestedTokens: 1
```

The following example configures a KeyResolver in Java:

Example 41. Config.java

```
@Bean
KeyResolver userKeyResolver() {
    return exchange ->
Mono.just(exchange.getRequest().getQueryParams().getFirst("user"));
}
```

This defines a request rate limit of 10 per user. A burst of 20 is allowed, but, in the next second, only 10 requests are available. The `KeyResolver` is a simple one that gets the `user` request parameter (note that this is not recommended for production).

You can also define a rate limiter as a bean that implements the `RateLimiter` interface. In configuration, you can reference the bean by name using SpEL. `#{@myRateLimiter}` is a SpEL expression that references a bean with named `myRateLimiter`. The following listing defines a rate limiter that uses the `KeyResolver` defined in the previous listing:

Example 42. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: requestratelimiter_route
        uri: https://example.org
        filters:
        - name: RequestRateLimiter
          args:
            rate-limiter: "#{@myRateLimiter}"
            key-resolver: "#{@userKeyResolver}"
```

74.12. The `RedirectTo` `GatewayFilter` Factory

The `RedirectTo` `GatewayFilter` factory takes two parameters, `status` and `url`. The `status` parameter should be a 300 series redirect HTTP code, such as 301. The `url` parameter should be a valid URL. This is the value of the `Location` header. For relative redirects, you should use `uri: no://op` as the uri of your route definition. The following listing configures a `RedirectTo` `GatewayFilter`:

Example 43. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: https://example.org
        filters:
        - RedirectTo=302, https://acme.org
```

This will send a status 302 with a `Location:https://acme.org` header to perform a redirect.

74.13. The `RemoveRequestHeader` GatewayFilter Factory

The `RemoveRequestHeader` `GatewayFilter` factory takes a `name` parameter. It is the name of the header to be removed. The following listing configures a `RemoveRequestHeader` `GatewayFilter`:

Example 44. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: removerequestheader_route
        uri: https://example.org
        filters:
        - RemoveRequestHeader=X-Request-Foo
```

This removes the `X-Request-Foo` header before it is sent downstream.

74.14. `RemoveResponseHeader` `GatewayFilter` Factory

The `RemoveResponseHeader` `GatewayFilter` factory takes a `name` parameter. It is the name of the header to be removed. The following listing configures a `RemoveResponseHeader` `GatewayFilter`:

Example 45. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: removeresponseheader_route
        uri: https://example.org
        filters:
        - RemoveResponseHeader=X-Response-Foo
```

This will remove the `X-Response-Foo` header from the response before it is returned to the gateway client.

To remove any kind of sensitive header, you should configure this filter for any routes for which you may want to do so. In addition, you can configure this filter once by using `spring.cloud.gateway.default-filters` and have it applied to all routes.

74.15. The `RemoveRequestParameter` `GatewayFilter` Factory

The `RemoveRequestParameter` `GatewayFilter` factory takes a `name` parameter. It is the name of the query parameter to be removed. The following example configures a `RemoveRequestParameter` `GatewayFilter`:

Example 46. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: removerequestparameter_route
        uri: https://example.org
        filters:
        - RemoveRequestParameter=red
```

This will remove the `red` parameter before it is sent downstream.

74.16. The `RewritePath` `GatewayFilter` Factory

The `RewritePath` `GatewayFilter` factory takes a path `regexp` parameter and a `replacement` parameter. This uses Java regular expressions for a flexible way to rewrite the request path. The following listing configures a `RewritePath` `GatewayFilter`:

Example 47. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: rewritepath_route
        uri: https://example.org
        predicates:
        - Path=/red/**
        filters:
        - RewritePath=/red(?&lt;segment&gt;/?.*), $\{segment}
```

For a request path of `/red/blue`, this sets the path to `/blue` before making the downstream request. Note that the `$` should be replaced with `$\` because of the YAML specification.

74.17. `RewriteLocationResponseHeader` `GatewayFilter` Factory

The `RewriteLocationResponseHeader` `GatewayFilter` factory modifies the value of the `Location` response header, usually to get rid of backend-specific details. It takes `stripVersionMode`, `locationHeaderName`, `hostValue`, and `protocolsRegex` parameters. The following listing configures a `RewriteLocationResponseHeader` `GatewayFilter`:

Example 48. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: rewritelocationresponseheader_route
        uri: http://example.org
        filters:
        - RewriteLocationResponseHeader=AS_IN_REQUEST, Location, ,
```

For example, for a request of `POST` `api.example.com/some/object/name`, the `Location` response header value of `object-service.prod.example.net/v2/some/object/id` is rewritten as `api.example.com/some/object/id`.

The `stripVersionMode` parameter has the following possible values: `NEVER_STRIP`, `AS_IN_REQUEST` (default), and `ALWAYS_STRIP`.

- `NEVER_STRIP`: The version is not stripped, even if the original request path contains no version.
- `AS_IN_REQUEST` The version is stripped only if the original request path contains no version.
- `ALWAYS_STRIP` The version is always stripped, even if the original request path contains version.

The `hostValue` parameter, if provided, is used to replace the `host:port` portion of the response `Location` header. If it is not provided, the value of the `Host` request header is used.

The `protocolsRegex` parameter must be a valid regex `String`, against which the protocol name is matched. If it is not matched, the filter does nothing. The default is `http|https|ftp|ftps`.

74.18. The `RewriteResponseHeader` `GatewayFilter` Factory

The `RewriteResponseHeader` `GatewayFilter` factory takes `name`, `regexp`, and `replacement` parameters. It uses Java regular expressions for a flexible way to rewrite the response header value. The following example configures a `RewriteResponseHeader` `GatewayFilter`:

Example 49. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: rewriteresponseheader_route
        uri: https://example.org
        filters:
        - RewriteResponseHeader=X-Response-Red, , password=[^&]+, password=***
```

For a header value of `/42?user=ford&password=omg!what&flag=true`, it is set to `/42?user=ford&password=***&flag=true` after making the downstream request. You must use `$\` to mean `$` because of the YAML specification.

74.19. The `SaveSession` `GatewayFilter` Factory

The `SaveSession` `GatewayFilter` factory forces a `WebSession::save` operation *before* forwarding the call downstream. This is of particular use when using something like Spring Session with a lazy data store and you need to ensure the session state has been saved before making the forwarded call. The following example configures a `SaveSession` `GatewayFilter`:

Example 50. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: save_session
        uri: https://example.org
        predicates:
        - Path=/foo/**
        filters:
        - SaveSession
```

If you integrate Spring Security with Spring Session and want to ensure security details have been forwarded to the remote process, this is critical.

74.20. The `SecureHeaders` `GatewayFilter` Factory

The `SecureHeaders` `GatewayFilter` factory adds a number of headers to the response, per the recommendation made in this blog post.

The following headers (shown with their default values) are added:

- `X-Xss-Protection:1 (mode=block)`
- `Strict-Transport-Security (max-age=631138519)`
- `X-Frame-Options (DENY)`
- `X-Content-Type-Options (nosniff)`
- `Referrer-Policy (no-referrer)`
- `Content-Security-Policy (default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; object-src 'none'; script-src https:; style-src 'self' https: 'unsafe-inline)'`
- `X-Download-Options (noopen)`
- `X-Permitted-Cross-Domain-Policies (none)`

To change the default values, set the appropriate property in the

`spring.cloud.gateway.filter.secure-headers` namespace. The following properties are available:

- `xss-protection-header`
- `strict-transport-security`
- `x-frame-options`
- `x-content-type-options`
- `referrer-policy`
- `content-security-policy`
- `x-download-options`
- `x-permitted-cross-domain-policies`

To disable the default values set the `spring.cloud.gateway.filter.secure-headers.disable` property with comma-separated values. The following example shows how to do so:

```
spring.cloud.gateway.filter.secure-headers.disable=x-frame-options,strict-
transport-security
```

The lowercase full name of the secure header needs to be used to disable it..

74.21. The `SetPath` `GatewayFilter` Factory

The `SetPath GatewayFilter` factory takes a path `template` parameter. It offers a simple way to manipulate the request path by allowing templated segments of the path. This uses the URI templates from Spring Framework. Multiple matching segments are allowed. The following example configures a `SetPath GatewayFilter`:

Example 51. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setpath_route
        uri: https://example.org
        predicates:
        - Path=/red/{segment}
        filters:
        - SetPath=/{segment}
```

For a request path of `/red/blue`, this sets the path to `/blue` before making the downstream request.

74.22. The `SetRequestHeader` `GatewayFilter` Factory

The `SetRequestHeader` `GatewayFilter` factory takes `name` and `value` parameters. The following listing configures a `SetRequestHeader` `GatewayFilter`:

Example 52. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setrequestheader_route
        uri: https://example.org
        filters:
        - SetRequestHeader=X-Request-Red, Blue
```

This `GatewayFilter` replaces (rather than adding) all headers with the given name. So, if the downstream server responded with a `X-Request-Red:1234`, this would be replaced with `X-Request-Red:Blue`, which is what the downstream service would receive.

`SetRequestHeader` is aware of URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `SetRequestHeader` `GatewayFilter` that uses a variable:

Example 53. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setrequestheader_route
        uri: https://example.org
        predicates:
        - Host: {segment}.myhost.org
        filters:
        - SetRequestHeader=foo, bar-{segment}
```

74.23. The `SetResponseHeader` `GatewayFilter` Factory

The `SetResponseHeader` `GatewayFilter` factory takes `name` and `value` parameters. The following listing configures a `SetResponseHeader` `GatewayFilter`:

Example 54. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setresponseheader_route
        uri: https://example.org
        filters:
        - SetResponseHeader=X-Response-Red, Blue
```

This GatewayFilter replaces (rather than adding) all headers with the given name. So, if the downstream server responded with a `X-Response-Red:1234`, this is replaced with `X-Response-Red:Blue`, which is what the gateway client would receive.

`SetResponseHeader` is aware of URI variables used to match a path or host. URI variables may be used in the value and will be expanded at runtime. The following example configures an `SetResponseHeader GatewayFilter` that uses a variable:

Example 55. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setresponseheader_route
        uri: https://example.org
        predicates:
        - Host: {segment}.myhost.org
        filters:
        - SetResponseHeader=foo, bar-{segment}
```

74.24. The `SetStatus GatewayFilter` Factory

The `SetStatus GatewayFilter` factory takes a single parameter, `status`. It must be a valid Spring `HttpStatus`. It may be the integer value `404` or the string representation of the enumeration: `NOT_FOUND`. The following listing configures a `SetStatus GatewayFilter`:

Example 56. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setstatusstring_route
        uri: https://example.org
        filters:
        - SetStatus=BAD_REQUEST
      - id: setstatusint_route
        uri: https://example.org
        filters:
        - SetStatus=401
```

In either case, the HTTP status of the response is set to 401.

You can configure the `SetStatus GatewayFilter` to return the original HTTP status code from the proxied request in a header in the response. The header is added to the response if configured with the following property:

Example 57. application.yml

```
spring:
  cloud:
    gateway:
      set-status:
        original-status-header-name: original-http-status
```

74.25. The `StripPrefix GatewayFilter` Factory

The `StripPrefix GatewayFilter` factory takes one parameter, `parts`. The `parts` parameter indicates the number of parts in the path to strip from the request before sending it downstream. The following listing configures a `StripPrefix GatewayFilter`:

Example 58. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: nameRoot
        uri: https://nameservice
        predicates:
        - Path=/name/**
        filters:
        - StripPrefix=2
```

When a request is made through the gateway to `/name/blue/red`, the request made to `nameservice` looks like `nameservice/red`.

74.26. The Retry `GatewayFilter` Factory

The `Retry GatewayFilter` factory supports the following parameters:

- `retries`: The number of retries that should be attempted.
- `statuses`: The HTTP status codes that should be retried, represented by using `org.springframework.http.HttpStatus`.
- `methods`: The HTTP methods that should be retried, represented by using `org.springframework.http.HttpMethod`.
- `series`: The series of status codes to be retried, represented by using `org.springframework.http.HttpStatus.Series`.
- `exceptions`: A list of thrown exceptions that should be retried.
- `backoff`: The configured exponential backoff for the retries. Retries are performed after a backoff interval of `firstBackoff * (factor ^ n)`, where `n` is the iteration. If `maxBackoff` is configured, the maximum backoff applied is limited to `maxBackoff`. If `basedOnPreviousValue` is true, the backoff is calculated byusing `prevBackoff * factor`.

The following defaults are configured for `Retry` filter, if enabled:

- `retries`: Three times
- `series`: 5XX series
- `methods`: GET method
- `exceptions`: `IOException` and `TimeoutException`
- `backoff`: disabled

The following listing configures a Retry `GatewayFilter`:

Example 59. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: retry_test
        uri: http://localhost:8080/flakey
        predicates:
        - Host=*.retry.com
        filters:
        - name: Retry
          args:
            retries: 3
            statuses: BAD_GATEWAY
            methods: GET,POST
            backoff:
              firstBackoff: 10ms
              maxBackoff: 50ms
              factor: 2
              basedOnPreviousValue: false
```

When using the retry filter with a `forward:` prefixed URL, the target endpoint should be written carefully so that, in case of an error, it does not do anything that could result in a response being sent to the client and committed. For example, if the target endpoint is an annotated controller, the target controller method should not return `ResponseEntity` with an error status code. Instead, it should throw an `Exception` or signal an error (for example, through a `Mono.error(ex)` return value), which the retry filter can be configured to handle by retrying.

When using the retry filter with any HTTP method with a body, the body will be cached and the gateway will become memory constrained. The body is cached in a request attribute defined by `ServerWebExchangeUtils.CACHED_REQUEST_BODY_ATTR`. The type of the object is a `org.springframework.core.io.buffer.DataBuffer`.

74.27. The `RequestSize` `GatewayFilter` Factory

When the request size is greater than the permissible limit, the `RequestSize` `GatewayFilter` factory can restrict a request from reaching the downstream service. The filter takes a `maxSize` parameter. The `maxSize is a `DataSize` type, so values can be defined as a number followed by an optional `DataUnit` suffix such as 'KB' or 'MB'. The default is 'B' for bytes. It is the permissible size limit of the request defined in bytes. The following listing configures a `RequestSize` `GatewayFilter`:

Example 60. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: request_size_route
        uri: http://localhost:8080/upload
        predicates:
        - Path=/upload
        filters:
        - name: RequestSize
          args:
            maxSize: 5000000
```

The `RequestSize` `GatewayFilter` factory sets the response status as `413 Payload Too Large` with an additional header `errorMessage` when the request is rejected due to size. The following example shows such an `errorMessage`:

```
errorMessage` : `Request size is larger than permissible limit. Request size is
6.0 MB where permissible limit is 5.0 MB
```

The default request size is set to five MB if not provided as a filter argument in the route definition.

74.28. The `SetRequestHost` `GatewayFilter` Factory

There are certain situation when the host header may need to be overridden. In this situation, the `SetRequestHost` `GatewayFilter` factory can replace the existing host header with a specified vaue. The filter takes a `host` parameter. The following listing configures a `SetRequestHost` `GatewayFilter`:

Example 61. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: set_request_host_header_route
        uri: http://localhost:8080/headers
        predicates:
        - Path=/headers
        filters:
        - name: SetRequestHost
          args:
            host: example.org
```

The `SetRequestHost` `GatewayFilter` factory replaces the value of the host header with `example.org`.

74.29. Modify a Request Body `GatewayFilter` Factory

You can use the `ModifyRequestBody` filter filter to modify the request body before it is sent downstream by the gateway.

This filter can be configured only by using the Java DSL.

The following listing shows how to modify a request body `GatewayFilter`:

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("rewrite_request_obj", r -> r.host("*.rewriterequestobj.org")
            .filters(f -> f.prefixPath("/httpbin")
                .modifyRequestBody(String.class, Hello.class,
MediaType.APPLICATION_JSON_VALUE,
                    (exchange, s) -> return Mono.just(new
Hello(s.toUpperCase())))).uri(uri))
        .build();
}

static class Hello {
    String message;

    public Hello() { }

    public Hello(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

74.30. Modify a Response Body `GatewayFilter` Factory

You can use the `ModifyResponseBody` filter to modify the response body before it is sent back to the client.

This filter can be configured only by using the Java DSL.

The following listing shows how to modify a response body `GatewayFilter`:

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("rewrite_response_upper", r -> r.host("*.rewriteresponseupper.org")
            .filters(f -> f.prefixPath("/httpbin")
                .modifyResponseBody(String.class, String.class,
                    (exchange, s) -> Mono.just(s.toUpperCase()))).uri(uri))
        .build();
}
```

74.31. Default Filters

To add a filter and apply it to all routes, you can use `spring.cloud.gateway.default-filters`. This property takes a list of filters. The following listing defines a set of default filters:

Example 62. application.yml

```
spring:
  cloud:
    gateway:
      default-filters:
      - AddResponseHeader=X-Response-Default-Red, Default-Blue
      - PrefixPath=/httpbin
```

Chapter 75. Global Filters

The `GlobalFilter` interface has the same signature as `GatewayFilter`. These are special filters that are conditionally applied to all routes.

This interface and its usage are subject to change in future milestone releases.

75.1. Combined Global Filter and `GatewayFilter` Ordering

When a request matches a route, the filtering web handler adds all instances of `GlobalFilter` and all route-specific instances of `GatewayFilter` to a filter chain. This combined filter chain is sorted by the `org.springframework.core.Ordered` interface, which you can set by implementing the `getOrder()` method.

As Spring Cloud Gateway distinguishes between “pre” and “post” phases for filter logic execution (see How it Works), the filter with the highest precedence is the first in the “pre”-phase and the last in the “post”-phase.

The following listing configures a filter chain:

Example 63. ExampleConfiguration.java

```
@Bean
public GlobalFilter customFilter() {
    return new CustomGlobalFilter();
}

public class CustomGlobalFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain)
{
        log.info("custom global filter");
        return chain.filter(exchange);
    }

    @Override
    public int getOrder() {
        return -1;
    }
}
```

75.2. Forward Routing Filter

The `ForwardRoutingFilter` looks for a URI in the exchange attribute `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`. If the URL has a `forward` scheme (such as `forward:///localendpoint`), it uses the Spring `DispatcherHandler` to handle the request. The path part of the request URL is overridden with the path in the forward URL. The unmodified original URL is appended to the list in the `ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR` attribute.

75.3. The `LoadBalancerClient` Filter

The `LoadBalancerClientFilter` looks for a URI in the exchange attribute named `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`. If the URL has a scheme of `lb` (such as `lb://myservice`), it uses the Spring Cloud `LoadBalancerClient` to resolve the name (`myservice` in this case) to an actual host and port and replaces the URI in the same attribute. The unmodified original URL is appended to the list in the `ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR` attribute. The filter also looks in the `ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR` attribute to see if it equals `lb`. If so, the same rules apply. The following listing configures a `LoadBalancerClientFilter`:

Example 64. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: myRoute
        uri: lb://service
        predicates:
        - Path=/service/**
```

By default, when a service instance cannot be found in the `LoadBalancer`, a `503` is returned. You can configure the Gateway to return a `404` by setting `spring.cloud.gateway.loadbalancer.use404=true`.

The `isSecure` value of the `ServiceInstance` returned from the `LoadBalancer` overrides the scheme specified in the request made to the Gateway. For example, if the request comes into the Gateway over `HTTPS` but the `ServiceInstance` indicates it is not secure, the downstream request is made over `HTTP`. The opposite situation can also apply. However, if `GATEWAY_SCHEME_PREFIX_ATTR` is specified for the route in the Gateway configuration, the prefix is stripped and the resulting scheme from the route URL overrides the `ServiceInstance` configuration.

LoadBalancerClientFilter uses a blocking ribbon LoadBalancerClient under the hood. We suggest you use ReactiveLoadBalancerClientFilter instead. You can switch to it by setting the value of the spring.cloud.loadbalancer.ribbon.enabled to false.

75.4. The ReactiveLoadBalancerClientFilter

The ReactiveLoadBalancerClientFilter looks for a URI in the exchange attribute named ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR. If the URL has a lb scheme (such as lb://myservice), it uses the Spring Cloud ReactorLoadBalancer to resolve the name (myservice in this example) to an actual host and port and replaces the URI in the same attribute. The unmodified original URL is appended to the list in the ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR attribute. The filter also looks in the ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR attribute to see if it equals lb. If so, the same rules apply. The following listing configures a ReactiveLoadBalancerClientFilter:

Example 65. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: myRoute
        uri: lb://service
        predicates:
        - Path=/service/**
```

By default, when a service instance cannot be found by the ReactorLoadBalancer, a 503 is returned. You can configure the gateway to return a 404 by setting spring.cloud.gateway.loadbalancer.use404=true.

The isSecure value of the ServiceInstance returned from the ReactiveLoadBalancerClientFilter overrides the scheme specified in the request made to the Gateway. For example, if the request comes into the Gateway over HTTPS but the ServiceInstance indicates it is not secure, the downstream request is made over HTTP. The opposite situation can also apply. However, if GATEWAY_SCHEME_PREFIX_ATTR is specified for the route in the Gateway configuration, the prefix is stripped and the resulting scheme from the route URL overrides the ServiceInstance configuration.

75.5. The Netty Routing Filter

The Netty routing filter runs if the URL located in the ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR exchange attribute has a http or https scheme. It uses the Netty HttpClient to make the downstream proxy request. The response is put in the

`ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR` exchange attribute for use in a later filter. (There is also an experimental `WebClientHttpRoutingFilter` that performs the same function but does not require Netty.)

75.6. The Netty Write Response Filter

The `NettyWriteResponseFilter` runs if there is a Netty `HttpClientResponse` in the `ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR` exchange attribute. It runs after all other filters have completed and writes the proxy response back to the gateway client response. (There is also an experimental `WebClientWriteResponseFilter` that performs the same function but does not require Netty.)

75.7. The `RouteToRequestUrl` Filter

If there is a `Route` object in the `ServerWebExchangeUtils.GATEWAY_ROUTE_ATTR` exchange attribute, the `RouteToRequestUrlFilter` runs. It creates a new URI, based off of the request URI but updated with the URI attribute of the `Route` object. The new URI is placed in the `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute`.

If the URI has a scheme prefix, such as `lb:ws://serviceid`, the `lb` scheme is stripped from the URI and placed in the `ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR` for use later in the filter chain.

75.8. The Websocket Routing Filter

If the URL located in the `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute has a `ws` or `wss` scheme, the websocket routing filter runs. It uses the Spring WebSocket infrastructure to forward the websocket request downstream.

You can load-balance websockets by prefixing the URI with `lb`, such as `lb:ws://serviceid`.

If you use SockJS as a fallback over normal HTTP, you should configure a normal HTTP route as well as the websocket Route.

The following listing configures a websocket routing filter:

Example 66. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      # SockJS route
      - id: websocket_sockjs_route
        uri: http://localhost:3001
        predicates:
        - Path=/websocket/info/**
      # Normal Websocket route
      - id: websocket_route
        uri: ws://localhost:3001
        predicates:
        - Path=/websocket/**
```

75.9. The Gateway Metrics Filter

To enable gateway metrics, add spring-boot-starter-actuator as a project dependency. Then, by default, the gateway metrics filter runs as long as the property `spring.cloud.gateway.metrics.enabled` is not set to `false`. This filter adds a timer metric named `gateway.requests` with the following tags:

- `routeId`: The route ID.
- `routeUri`: The URI to which the API is routed.
- `outcome`: The outcome, as classified by HttpStatus.Series.
- `status`: The HTTP status of the request returned to the client.
- `httpStatusCode`: The HTTP Status of the request returned to the client.
- `httpMethod`: The HTTP method used for the request.

These metrics are then available to be scraped from `/actuator/metrics/gateway.requests` and can be easily integrated with Prometheus to create a Grafana dashboard.

To enable the prometheus endpoint, add `micrometer-registry-prometheus` as a project dependency.

75.10. Marking An Exchange As Routed

After the gateway has routed a `ServerWebExchange`, it marks that exchange as “routed” by adding `gatewayAlreadyRouted` to the exchange attributes. Once a request has been marked as routed, other routing filters will not route the request again, essentially skipping the filter. There are convenience methods that you can use to mark an exchange as routed or check if an exchange has already been routed.

* `ServerWebExchangeUtils.isAlreadyRouted` takes a `ServerWebExchange` object and checks if it has been “routed”.
* `ServerWebExchangeUtils.setAlreadyRouted` takes a `ServerWebExchange` object and marks it as “routed”.

Chapter 76. HttpHeadersFilters

HttpHeadersFilters are applied to requests before sending them downstream, such as in the `NettyRoutingFilter`.

76.1. Forwarded Headers Filter

The `Forwarded` Headers Filter creates a `Forwarded` header to send to the downstream service. It adds the `Host` header, scheme and port of the current request to any existing `Forwarded` header.

76.2. RemoveHopByHop Headers Filter

The `RemoveHopByHop` Headers Filter removes headers from forwarded requests. The default list of headers that is removed comes from the IETF.

The default removed headers are:

- Connection
- Keep-Alive
- Proxy-Authenticate
- Proxy-Authorization
- TE
- Trailer
- Transfer-Encoding
- Upgrade

To change this, set the `spring.cloud.gateway.filter.remove-hop-by-hop.headers` property to the list of header names to remove.

76.3. XForwarded Headers Filter

The `XForwarded` Headers Filter creates various a `X-Forwarded-*` headers to send to the downstream service. It users the `Host` header, scheme, port and path of the current request to create the various headers.

Creating of individual headers can be controlled by the following boolean properties (defaults to true):

- `spring.cloud.gateway.x-forwarded.for-enabled`
- `spring.cloud.gateway.x-forwarded.host-enabled`
- `spring.cloud.gateway.x-forwarded.port-enabled`
- `spring.cloud.gateway.x-forwarded.proto-enabled`
- `spring.cloud.gateway.x-forwarded.prefix-enabled`

Appending multiple headers can be controlled by the following boolean properties (defaults to

true):

- `spring.cloud.gateway.x-forwarded.for-append`
- `spring.cloud.gateway.x-forwarded.host-append`
- `spring.cloud.gateway.x-forwarded.port-append`
- `spring.cloud.gateway.x-forwarded.proto-append`
- `spring.cloud.gateway.x-forwarded.prefix-append`

Chapter 77. TLS and SSL

The gateway can listen for requests on HTTPS by following the usual Spring server configuration. The following example shows how to do so:

Example 67. application.yml

```
server:
  ssl:
    enabled: true
    key-alias: scg
    key-store-password: scg1234
    key-store: classpath:scg-keystore.p12
    key-store-type: PKCS12
```

You can route gateway routes to both HTTP and HTTPS backends. If you are routing to an HTTPS backend, you can configure the gateway to trust all downstream certificates with the following configuration:

Example 68. application.yml

```
spring:
  cloud:
    gateway:
      httpclient:
        ssl:
          useInsecureTrustManager: true
```

Using an insecure trust manager is not suitable for production. For a production deployment, you can configure the gateway with a set of known certificates that it can trust with the following configuration:

Example 69. application.yml

```
spring:
  cloud:
    gateway:
      httpclient:
        ssl:
          trustedX509Certificates:
          - cert1.pem
          - cert2.pem
```

If the Spring Cloud Gateway is not provisioned with trusted certificates, the default trust store is used (which you can override by setting the `javax.net.ssl.trustStore` system property).

77.1. TLS Handshake

The gateway maintains a client pool that it uses to route to backends. When communicating over HTTPS, the client initiates a TLS handshake. A number of timeouts are associated with this handshake. You can configure these timeouts can be configured (defaults shown) as follows:

Example 70. application.yml

```yaml
spring:
  cloud:
    gateway:
      httpclient:
        ssl:
          handshake-timeout-millis: 10000
          close-notify-flush-timeout-millis: 3000
          close-notify-read-timeout-millis: 0
```

Chapter 78. Configuration

Configuration for Spring Cloud Gateway is driven by a collection of `RouteDefinitionLocator` instances. The following listing shows the definition of the `RouteDefinitionLocator` interface:

Example 71. RouteDefinitionLocator.java

```
public interface RouteDefinitionLocator {
    Flux<RouteDefinition> getRouteDefinitions();
}
```

By default, a `PropertiesRouteDefinitionLocator` loads properties by using Spring Boot's `@ConfigurationProperties` mechanism.

The earlier configuration examples all use a shortcut notation that uses positional arguments rather than named ones. The following two examples are equivalent:

Example 72. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: setstatus_route
        uri: https://example.org
        filters:
        - name: SetStatus
          args:
            status: 401
      - id: setstatusshortcut_route
        uri: https://example.org
        filters:
        - SetStatus=401
```

For some usages of the gateway, properties are adequate, but some production use cases benefit from loading configuration from an external source, such as a database. Future milestone versions will have `RouteDefinitionLocator` implementations based off of Spring Data Repositories, such as Redis, MongoDB, and Cassandra.

Chapter 79. Route Metadata Configuration

You can configure additional parameters for each route by using metadata, as follows:

Example 73. application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: route_with_metadata
        uri: https://example.org
        metadata:
          optionName: "OptionValue"
          compositeObject:
            name: "value"
          iAmNumber: 1
```

You could acquire all metadata properties from an exchange, as follows:

```
Route route = exchange.getAttribute(GATEWAY_ROUTE_ATTR);
// get all metadata properties
route.getMetadata();
// get a single metadata property
route.getMetadata(someKey);
```

Chapter 80. Http timeouts configuration

Http timeouts (response and connect) can be configured for all routes and overridden for each specific route.

80.1. Global timeouts

To configure Global http timeouts:
`connect-timeout` must be specified in milliseconds.
`response-timeout` must be specified as a java.time.Duration

global http timeouts example

```
spring:
  cloud:
    gateway:
      httpclient:
        connect-timeout: 1000
        response-timeout: 5s
```

80.2. Per-route timeouts

To configure per-route timeouts:
`connect-timeout` must be specified in milliseconds.
`response-timeout` must be specified in milliseconds.

per-route http timeouts configuration via configuration

```
      - id: per_route_timeouts
        uri: https://example.org
        predicates:
          - name: Path
            args:
              pattern: /delay/{timeout}
        metadata:
          response-timeout: 200
          connect-timeout: 200
```

per-route timeouts configuration using Java DSL

```
import static
org.springframework.cloud.gateway.support.RouteMetadataUtils.CONNECT_TIMEOUT_ATTR;
import static
org.springframework.cloud.gateway.support.RouteMetadataUtils.RESPONSE_TIMEOUT_ATTR;

      @Bean
      public RouteLocator customRouteLocator(RouteLocatorBuilder routeBuilder){
         return routeBuilder.routes()
               .route("test1", r -> {
                  return r.host("*.somehost.org").and().path("/somepath")
                        .filters(f -> f.addRequestHeader("header1", "header-value-1"))
                        .uri("http://someuri")
                        .metadata(RESPONSE_TIMEOUT_ATTR, 200)
                        .metadata(CONNECT_TIMEOUT_ATTR, 200);
               })
               .build();
      }
```

80.3. Fluent Java Routes API

To allow for simple configuration in Java, the `RouteLocatorBuilder` bean includes a fluent API. The following listing shows how it works:

Example 74. GatewaySampleApplication.java

```
// static imports from GatewayFilters and RoutePredicates
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder,
ThrottleGatewayFilterFactory throttle) {
    return builder.routes()
            .route(r -> r.host("**.abc.org").and().path("/image/png")
                .filters(f ->
                        f.addResponseHeader("X-TestHeader", "foobar"))
                .uri("http://httpbin.org:80")
            )
            .route(r -> r.path("/image/webp")
                .filters(f ->
                        f.addResponseHeader("X-AnotherHeader", "baz"))
                .uri("http://httpbin.org:80")
                .metadata("key", "value")
            )
            .route(r -> r.order(-1)
                .host("**.throttle.org").and().path("/get")
                .filters(f -> f.filter(throttle.apply(1,
                        1,
                        10,
                        TimeUnit.SECONDS)))
                .uri("http://httpbin.org:80")
                .metadata("key", "value")
            )
            .build();
}
```

This style also allows for more custom predicate assertions. The predicates defined by `RouteDefinitionLocator` beans are combined using logical `and`. By using the fluent Java API, you can use the `and()`, `or()`, and `negate()` operators on the `Predicate` class.

80.4. The `DiscoveryClient` Route Definition Locator

You can configure the gateway to create routes based on services registered with a `DiscoveryClient` compatible service registry.

To enable this, set `spring.cloud.gateway.discovery.locator.enabled=true` and make sure a `DiscoveryClient` implementation (such as Netflix Eureka, Consul, or Zookeeper) is on the classpath and enabled.

80.4.1. Configuring Predicates and Filters For `DiscoveryClient` Routes

By default, the gateway defines a single predicate and filter for routes created with a `DiscoveryClient`.

The default predicate is a path predicate defined with the pattern `/serviceId/**`, where `serviceId` is the ID of the service from the `DiscoveryClient`.

The default filter is a rewrite path filter with the regex `/serviceId/(?<remaining>.*)` and the replacement `/${remaining}`. This strips the service ID from the path before the request is sent downstream.

If you want to customize the predicates or filters used by the `DiscoveryClient` routes, set `spring.cloud.gateway.discovery.locator.predicates[x]` and `spring.cloud.gateway.discovery.locator.filters[y]`. When doing so, you need to make sure to include the default predicate and filter shown earlier, if you want to retain that functionality. The following example shows what this looks like:

Example 75. application.properties

```
spring.cloud.gateway.discovery.locator.predicates[0].name: Path
spring.cloud.gateway.discovery.locator.predicates[0].args[pattern]:
"'/'+serviceId+'/**'"
spring.cloud.gateway.discovery.locator.predicates[1].name: Host
spring.cloud.gateway.discovery.locator.predicates[1].args[pattern]: "'**.foo.com'"
spring.cloud.gateway.discovery.locator.filters[0].name: Hystrix
spring.cloud.gateway.discovery.locator.filters[0].args[name]: serviceId
spring.cloud.gateway.discovery.locator.filters[1].name: RewritePath
spring.cloud.gateway.discovery.locator.filters[1].args[regexp]: "'/' + serviceId +
'/(?<remaining>.*)'"
spring.cloud.gateway.discovery.locator.filters[1].args[replacement]:
"'/${remaining}'"
```

Chapter 81. Reactor Netty Access Logs

To enable Reactor Netty access logs, set `-Dreactor.netty.http.server.accessLogEnabled=true`.

It must be a Java System Property, not a Spring Boot property.

You can configure the logging system to have a separate access log file. The following example creates a Logback configuration:

Example 76. logback.xml

```
    <appender name="accessLog" class="ch.qos.logback.core.FileAppender">
        <file>access_log.log</file>
        <encoder>
            <pattern>%msg%n</pattern>
        </encoder>
    </appender>
    <appender name="async" class="ch.qos.logback.classic.AsyncAppender">
        <appender-ref ref="accessLog" />
    </appender>

    <logger name="reactor.netty.http.server.AccessLog" level="INFO"
additivity="false">
        <appender-ref ref="async"/>
    </logger>
```

Chapter 82. CORS Configuration

You can configure the gateway to control CORS behavior. The “global” CORS configuration is a map of URL patterns to Spring Framework `CorsConfiguration`. The following example configures CORS:

Example 77. application.yml

```
spring:
  cloud:
    gateway:
      globalcors:
        cors-configurations:
          '[/**]':
            allowedOrigins: "https://docs.spring.io"
            allowedMethods:
            - GET
```

In the preceding example, CORS requests are allowed from requests that originate from `docs.spring.io` for all GET requested paths.

To provide the same CORS configuration to requests that are not handled by some gateway route predicate, set the `spring.cloud.gateway.globalcors.add-to-simple-url-handler-mapping` property to `true`. This is useful when you try to support CORS preflight requests and your route predicate does not evalute to `true` because the HTTP method is `options`.

Chapter 83. Actuator API

The `/gateway` actuator endpoint lets you monitor and interact with a Spring Cloud Gateway application. To be remotely accessible, the endpoint has to be enabled and exposed over HTTP or JMX in the application properties. The following listing shows how to do so:

Example 78. application.properties

```
management.endpoint.gateway.enabled=true # default value
management.endpoints.web.exposure.include=gateway
```

83.1. Verbose Actuator Format

A new, more verbose format has been added to Spring Cloud Gateway. It adds more detail to each route, letting you view the predicates and filters associated with each route along with any configuration that is available. The following example configures `/actuator/gateway/routes`:

```
[
  {
    "predicate": "(Hosts: [**.addrequestheader.org] && Paths: [/headers], match
trailing slash: true)",
    "route_id": "add_request_header_test",
    "filters": [
      "[[AddResponseHeader X-Response-Default-Foo = 'Default-Bar'], order = 1]",
      "[[AddRequestHeader X-Request-Foo = 'Bar'], order = 1]",
      "[[PrefixPath prefix = '/httpbin'], order = 2]"
    ],
    "uri": "lb://testservice",
    "order": 0
  }
]
```

This feature is enabled by default. To disable it, set the following property:

Example 79. application.properties

```
spring.cloud.gateway.actuator.verbose.enabled=false
```

This will default to `true` in a future release.

83.2. Retrieving Route Filters

This section details how to retrieve route filters, including:

- Global Filters
- [gateway-route-filters]

83.2.1. Global Filters

To retrieve the global filters applied to all routes, make a `GET` request to `/actuator/gateway/globalfilters`. The resulting response is similar to the following:

```
{
  "org.springframework.cloud.gateway.filter.LoadBalancerClientFilter@77856cc5":
10100,
  "org.springframework.cloud.gateway.filter.RouteToRequestUrlFilter@4f6fd101":
10000,
  "org.springframework.cloud.gateway.filter.NettyWriteResponseFilter@32d22650":
-1,
  "org.springframework.cloud.gateway.filter.ForwardRoutingFilter@106459d9":
2147483647,
  "org.springframework.cloud.gateway.filter.NettyRoutingFilter@1fbd5e0":
2147483647,
  "org.springframework.cloud.gateway.filter.ForwardPathFilter@33a71d23": 0,
  "org.springframework.cloud.gateway.filter.AdaptCachedBodyGlobalFilter@135064ea":
2147483637,
  "org.springframework.cloud.gateway.filter.WebsocketRoutingFilter@23c05889":
2147483646
}
```

The response contains the details of the global filters that are in place. For each global filter, there is a string representation of the filter object (for example, `org.springframework.cloud.gateway.filter.LoadBalancerClientFilter@77856cc5`) and the corresponding order in the filter chain.}

83.2.2. Route Filters

To retrieve the `GatewayFilter` factories applied to routes, make a `GET` request to `/actuator/gateway/routefilters`. The resulting response is similar to the following:

```
{
  "[AddRequestHeaderGatewayFilterFactory@570ed9c configClass =
AbstractNameValueGatewayFilterFactory.NameValueConfig]": null,
  "[SecureHeadersGatewayFilterFactory@fceab5d configClass = Object]": null,
  "[SaveSessionGatewayFilterFactory@4449b273 configClass = Object]": null
}
```

The response contains the details of the `GatewayFilter` factories applied to any particular route. For each factory there is a string representation of the corresponding object (for example, `[SecureHeadersGatewayFilterFactory@fceab5d configClass = Object]`). Note that the `null` value is due to an incomplete implementation of the endpoint controller, because it tries to set the order of the object in the filter chain, which does not apply to a `GatewayFilter` factory object.

83.3. Refreshing the Route Cache

To clear the routes cache, make a `POST` request to `/actuator/gateway/refresh`. The request returns a 200 without a response body.

83.4. Retrieving the Routes Defined in the Gateway

To retrieve the routes defined in the gateway, make a `GET` request to `/actuator/gateway/routes`. The resulting response is similar to the following:

```
[{
  "route_id": "first_route",
  "route_object": {
    "predicate":
"org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory$$La
mbda$432/1736826640@1e9d7e7d",
    "filters": [

"OrderedGatewayFilter{delegate=org.springframework.cloud.gateway.filter.factory.Pr
eserveHostHeaderGatewayFilterFactory$$Lambda$436/674480275@6631ef72, order=0}"
    ]
  },
  "order": 0
},
{
  "route_id": "second_route",
  "route_object": {
    "predicate":
"org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory$$La
mbda$432/1736826640@cd8d298",
    "filters": []
  },
  "order": 0
}]
```

The response contains the details of all the routes defined in the gateway. The following table describes the structure of each element (each is a route) of the response:

Path	Type	Description
`route_id`	String	The route ID.
`route_object.predicate`	Object	The route predicate.
`route_object.filters`	Array	The `GatewayFilter` factories applied to the route.
`order`	Number	The route order.

83.5. Retrieving Information about a Particular Route

To retrieve information about a single route, make a `GET` request to `/actuator/gateway/routes/{id}` (for example, `/actuator/gateway/routes/first_route`). The resulting response is similar to the following:

```
{
  "id": "first_route",
  "predicates": [{
    "name": "Path",
    "args": {"_genkey_0":"/first"}
  }],
  "filters": [],
  "uri": "https://www.uri-destination.org",
  "order": 0
}]
```

The following table describes the structure of the response:

Path	Type	Description
`id`	String	The route ID.
`predicates`	Array	The collection of route predicates. Each item defines the name and the arguments of a given predicate.
`filters`	Array	The collection of filters applied to the route.
`uri`	String	The destination URI of the route.
`order`	Number	The route order.

83.6. Creating and Deleting a Particular Route

To create a route, make a `POST` request to `/gateway/routes/{id_route_to_create}` with a JSON body that specifies the fields of the route (see Retrieving Information about a Particular Route).

To delete a route, make a `DELETE` request to `/gateway/routes/{id_route_to_delete}`.

83.7. Recap: The List of All endpoints

The folloiwng table below summarizes the Spring Cloud Gateway actuator endpoints (note that each endpoint has `/actuator/gateway` as the base-path):

ID	HTTP Method	Description
`globalfilters`	GET	Displays the list of global filters applied to the routes.
`routefilters`	GET	Displays the list of `GatewayFilter` factories applied to a particular route.
`refresh`	POST	Clears the routes cache.
`routes`	GET	Displays the list of routes defined in the gateway.

ID	HTTP Method	Description
`routes/{id}`	GET	Displays information about a particular route.
`routes/{id}`	POST	Adds a new route to the gateway.
`routes/{id}`	DELETE	Removes an existing route from the gateway.

Chapter 84. Troubleshooting

This section covers common problems that may arise when you use Spring Cloud Gateway.

84.1. Log Levels

The following loggers may contain valuable troubleshooting information at the `DEBUG` and `TRACE` levels:

- `org.springframework.cloud.gateway`
- `org.springframework.http.server.reactive`
- `org.springframework.web.reactive`
- `org.springframework.boot.autoconfigure.web`
- `reactor.netty`
- `redisratelimiter`

84.2. Wiretap

The Reactor Netty `HttpClient` and `HttpServer` can have wiretap enabled. When combined with setting the `reactor.netty` log level to `DEBUG` or `TRACE`, it enables the logging of information, such as headers and bodies sent and received across the wire. To enable wiretap, set `spring.cloud.gateway.httpserver.wiretap=true` or `spring.cloud.gateway.httpclient.wiretap=true` for the `HttpServer` and `HttpClient`, respectively.

Chapter 85. Developer Guide

These are basic guides to writing some custom components of the gateway.

85.1. Writing Custom Route Predicate Factories

In order to write a Route Predicate you will need to implement `RoutePredicateFactory`. There is an abstract class called `AbstractRoutePredicateFactory` which you can extend.

MyRoutePredicateFactory.java

```
public class MyRoutePredicateFactory extends
AbstractRoutePredicateFactory<HeaderRoutePredicateFactory.Config> {

    public MyRoutePredicateFactory() {
        super(Config.class);
    }

    @Override
    public Predicate<ServerWebExchange> apply(Config config) {
        // grab configuration from Config object
        return exchange -> {
            //grab the request
            ServerHttpRequest request = exchange.getRequest();
            //take information from the request to see if it
            //matches configuration.
            return matches(config, request);
        };
    }

    public static class Config {
        //Put the configuration properties for your filter here
    }

}
```

85.2. Writing Custom GatewayFilter Factories

To write a `GatewayFilter`, you must implement `GatewayFilterFactory`. You can extend an abstract class called `AbstractGatewayFilterFactory`. The following examples show how to do so:

Example 80. PreGatewayFilterFactory.java

```
public class PreGatewayFilterFactory extends
AbstractGatewayFilterFactory<PreGatewayFilterFactory.Config> {

    public PreGatewayFilterFactory() {
        super(Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        // grab configuration from Config object
        return (exchange, chain) -> {
            //If you want to build a "pre" filter you need to manipulate the
            //request before calling chain.filter
            ServerHttpRequest.Builder builder = exchange.getRequest().mutate();
            //use builder to manipulate the request
            return
chain.filter(exchange.mutate().request(builder.build()).build());
        };
    }

    public static class Config {
        //Put the configuration properties for your filter here
    }

}
```

PostGatewayFilterFactory.java

```
public class PostGatewayFilterFactory extends
AbstractGatewayFilterFactory<PostGatewayFilterFactory.Config> {

    public PostGatewayFilterFactory() {
        super(Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        // grab configuration from Config object
        return (exchange, chain) -> {
            return chain.filter(exchange).then(Mono.fromRunnable(() -> {
                ServerHttpResponse response = exchange.getResponse();
                //Manipulate the response in some way
            }));
        };
    }

    public static class Config {
        //Put the configuration properties for your filter here
    }

}
```

85.2.1. Naming Custom Filters And References In Configuration

Custom filters class names should end in `GatewayFilterFactory`.

For example, to reference a filter named `Something` in configuration files, the filter must be in a class named `SomethingGatewayFilterFactory`.

It is possible to create a gateway filter named without the `GatewayFilterFactory` suffix, such as `class AnotherThing`. This filter could be referenced as `AnotherThing` in configuration files. This is **not** a supported naming convention and this syntax may be removed in future releases. Please update the filter name to be compliant.

85.3. Writing Custom Global Filters

To write a custom global filter, you must implement `GlobalFilter` interface. This applies the filter to all requests.

The following examples show how to set up global pre and post filters, respectively:

```
@Bean
public GlobalFilter customGlobalFilter() {
    return (exchange, chain) -> exchange.getPrincipal()
        .map(Principal::getName)
        .defaultIfEmpty("Default User")
        .map(userName -> {
          //adds header to proxied request
          exchange.getRequest().mutate().header("CUSTOM-REQUEST-HEADER",
userName).build();
          return exchange;
        })
        .flatMap(chain::filter);
}

@Bean
public GlobalFilter customGlobalPostFilter() {
    return (exchange, chain) -> chain.filter(exchange)
        .then(Mono.just(exchange))
        .map(serverWebExchange -> {
          //adds header to response
          serverWebExchange.getResponse().getHeaders().set("CUSTOM-RESPONSE-
HEADER",

HttpStatus.OK.equals(serverWebExchange.getResponse().getStatusCode()) ? "It
worked": "It did not work");
          return serverWebExchange;
        })
        .then();
}
```

Chapter 86. Building a Simple Gateway by Using Spring MVC or Webflux

The following describes an alternative style gateway. None of the prior documentation applies to what follows.

Spring Cloud Gateway provides a utility object called `ProxyExchange`. You can use it inside a regular Spring web handler as a method parameter. It supports basic downstream HTTP exchanges through methods that mirror the HTTP verbs. With MVC, it also supports forwarding to a local handler through the `forward()` method. To use the `ProxyExchange`, include the right module in your classpath (either `spring-cloud-gateway-mvc` or `spring-cloud-gateway-webflux`).

The following MVC example proxies a request to `/test` downstream to a remote server:

```
@RestController
@SpringBootApplication
public class GatewaySampleApplication {

    @Value("${remote.home}")
    private URI home;

    @GetMapping("/test")
    public ResponseEntity<?> proxy(ProxyExchange<byte[]> proxy) throws Exception {
        return proxy.uri(home.toString() + "/image/png").get();
    }

}
```

The following example does the same thing with Webflux:

```
@RestController
@SpringBootApplication
public class GatewaySampleApplication {

    @Value("${remote.home}")
    private URI home;

    @GetMapping("/test")
    public Mono<ResponseEntity<?>> proxy(ProxyExchange<byte[]> proxy) throws
Exception {
        return proxy.uri(home.toString() + "/image/png").get();
    }

}
```

Convenience methods on the `ProxyExchange` enable the handler method to discover and enhance the URI path of the incoming request. For example, you might want to extract the trailing elements of a path to pass them downstream:

```
@GetMapping("/proxy/path/**")
public ResponseEntity<?> proxyPath(ProxyExchange<byte[]> proxy) throws Exception {
  String path = proxy.path("/proxy/path/");
  return proxy.uri(home.toString() + "/foos/" + path).get();
}
```

All the features of Spring MVC and Webflux are available to gateway handler methods. As a result, you can inject request headers and query parameters, for instance, and you can constrain the incoming requests with declarations in the mapping annotation. See the documentation for `@RequestMapping` in Spring MVC for more details of those features.

You can add headers to the downstream response by using the `header()` methods on `ProxyExchange`.

You can also manipulate response headers (and anything else you like in the response) by adding a mapper to the `get()` method (and other methods). The mapper is a `Function` that takes the incoming `ResponseEntity` and converts it to an outgoing one.

First-class support is provided for “sensitive” headers (by default, `cookie` and `authorization`), which are not passed downstream, and for “proxy” (`x-forwarded-*`) headers.

Chapter 87. Configuration properties

To see the list of all Spring Cloud Gateway related configuration properties, see the appendix.

Spring Cloud GCP

João André Martins; Jisha Abubaker; Ray Tsang; Mike Eltsufin; Artem Bilan; Andreas Berger; Balint Pato; Chengyuan Zhao; Dmitry Solomakha; Elena Felder; Daniel Zou, Eddú Meléndez

Chapter 88. Introduction

The Spring Cloud GCP project makes the Spring Framework a first-class citizen of Google Cloud Platform (GCP).

Spring Cloud GCP lets you leverage the power and simplicity of the Spring Framework to:

- Publish and subscribe to Google Cloud Pub/Sub topics
- Configure Spring JDBC with a few properties to use Google Cloud SQL
- Map objects, relationships, and collections with Spring Data Cloud Spanner, Spring Data Cloud Datastore and Spring Data Reactive Repositories for Cloud Firestore
- Write and read from Spring Resources backed up by Google Cloud Storage
- Exchange messages with Spring Integration using Google Cloud Pub/Sub on the background
- Trace the execution of your app with Spring Cloud Sleuth and Google Stackdriver Trace
- Configure your app with Spring Cloud Config, backed up by the Google Runtime Configuration API
- Consume and produce Google Cloud Storage data via Spring Integration GCS Channel Adapters
- Use Spring Security via Google Cloud IAP
- Analyze your images for text, objects, and other content with Google Cloud Vision

Chapter 89. Getting Started

This section describes how to get up to speed with Spring Cloud GCP libraries.

89.1. Setting up Dependencies

All Spring Cloud GCP artifacts are made available through Maven Central. The following resources are provided to help you setup the libraries for your project:

- Maven Bill of Materials for dependency management
- Starter Dependencies for depending on Spring Cloud GCP modules

You may also consult our Github project to examine the code or build directly from source.

89.1.1. Bill of Materials

The Spring Cloud GCP Bill of Materials (BOM) contains the versions of all the dependencies it uses.

If you're a Maven user, adding the following to your pom.xml file will allow you omit any Spring Cloud GCP dependency version numbers from your configuration. Instead, the version of the BOM you're using determines the versions of the used dependencies.

```
<dependencyManagement>
   <dependencies>
       <dependency>
           <groupId>org.springframework.cloud</groupId>
           <artifactId>spring-cloud-gcp-dependencies</artifactId>
           <version>1.2.5.RELEASE</version>
           <type>pom</type>
           <scope>import</scope>
       </dependency>
   </dependencies>
</dependencyManagement>
```

See the sections in the README for selecting an available version and Maven repository.

In the following sections, it will be assumed you are using the Spring Cloud GCP BOM and the dependency snippets will not contain versions.

Gradle users can achieve the same kind of BOM experience using Spring's dependency-management-plugin Gradle plugin. For simplicity, the Gradle dependency snippets in the remainder of this document will also omit their versions.

89.1.2. Starter Dependencies

Spring Cloud GCP offers starter dependencies through Maven to easily depend on different modules of the library. Each starter contains all the dependencies and transitive dependencies needed to begin using their corresponding Spring Cloud GCP module.

For example, if you wish to write a Spring application with Cloud Pub/Sub, you would include the `spring-cloud-gcp-starter-pubsub` dependency in your project. You do **not** need to include the underlying `spring-cloud-gcp-pubsub` dependency, because the `starter` dependency includes it.

A summary of these artifacts are provided below.

Spring Cloud GCP Starter	Description	Maven Artifact Name
Core	Automatically configure authentication and Google project settings	org.springframework.cloud:spring-cloud-gcp-starter
Cloud Spanner	Provides integrations with Google Cloud Spanner	org.springframework.cloud:spring-cloud-gcp-starter-data-spanner
Cloud Datastore	Provides integrations with Google Cloud Datastore	org.springframework.cloud:spring-cloud-gcp-starter-data-datastore
Cloud Pub/Sub	Provides integrations with Google Cloud Pub/Sub	org.springframework.cloud:spring-cloud-gcp-starter-pubsub
Logging	Enables Stackdriver Logging	org.springframework.cloud:spring-cloud-gcp-starter-logging
SQL - MySQL	Cloud SQL integrations with MySQL	org.springframework.cloud:spring-cloud-gcp-starter-sql-mysql
SQL - PostgreSQL	Cloud SQL integrations with PostgreSQL	org.springframework.cloud:spring-cloud-gcp-starter-sql-postgresql
Storage	Provides integrations with Google Cloud Storage and Spring Resource	org.springframework.cloud:spring-cloud-gcp-starter-storage
Config	Enables usage of Google Runtime Configuration API as a Spring Cloud Config server	org.springframework.cloud:spring-cloud-gcp-starter-config
Trace	Enables instrumentation with Google Stackdriver Tracing	org.springframework.cloud:spring-cloud-gcp-starter-trace
Vision	Provides integrations with Google Cloud Vision	org.springframework.cloud:spring-cloud-gcp-starter-vision
Security - IAP	Provides a security layer over applications deployed to Google Cloud	org.springframework.cloud:spring-cloud-gcp-starter-security-iap

89.1.3. Spring Initializr

Spring Initializr is a tool which generates the scaffolding code for a new Spring Boot project. It handles the work of generating the Maven or Gradle build file so you do not have to manually add

the dependencies yourself.

Spring Initializr offers three modules from Spring Cloud GCP that you can use to generate your project.

- **GCP Support**: The GCP Support module contains auto-configuration support for every Spring Cloud GCP integration. Most of the autoconfiguration code is only enabled if the required dependency is added to your project.
- **GCP Messaging**: Google Cloud Pub/Sub integrations work out of the box.
- **GCP Storage**: Google Cloud Storage integrations work out of the box.

89.2. Learning Spring Cloud GCP

There are a variety of resources to help you learn how to use Spring Cloud GCP libraries.

89.2.1. Sample Applications

The easiest way to learn how to use Spring Cloud GCP is to consult the sample applications on Github. Spring Cloud GCP provides sample applications which demonstrate how to use every integration in the library. The table below highlights several samples of the most commonly used integrations in Spring Cloud GCP.

GCP Integration	Sample Application
Cloud Pub/Sub	spring-cloud-gcp-pubsub-sample
Cloud Spanner	spring-cloud-gcp-data-spanner-sample
Datastore	spring-cloud-gcp-data-datastore-sample
Cloud SQL (w/ MySQL)	spring-cloud-gcp-sql-mysql-sample
Cloud Storage	spring-cloud-gcp-storage-resource-sample
Stackdriver Logging	spring-cloud-gcp-logging-sample
Trace	spring-cloud-gcp-trace-sample
Cloud Vision	spring-cloud-gcp-vision-api-sample
Cloud Security - IAP	spring-cloud-gcp-security-iap-sample

Each sample application demonstrates how to use Spring Cloud GCP libraries in context and how to setup the dependencies for the project. The applications are fully functional and can be deployed to Google Cloud Platform as well. If you are interested, you may consult guides for deploying an application to AppEngine and to Google Kubernetes Engine.

89.2.2. Codelabs

For a more hands-on approach, there are several guides and codelabs to help you get up to speed. These guides provide step-by-step instructions for building an application using Spring Cloud GCP.

Some examples include:

- Deploy a Spring Boot app to App Engine
- Build a Kotlin Spring Boot app with Cloud SQL and Cloud Pub/Sub
- Build a Spring Boot application with Datastore
- Messaging with Spring Integration and Cloud Pub/Sub

The full collection of Spring codelabs can be found on the Google Developer Codelabs page.

Chapter 90. Spring Cloud GCP Core

Each Spring Cloud GCP module uses `GcpProjectIdProvider` and `CredentialsProvider` to get the GCP project ID and access credentials.

Spring Cloud GCP provides a Spring Boot starter to auto-configure the core components.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter'
}
```

90.1. Configuration

The following options may be configured with Spring Cloud core.

Name	Description	Required	Default value
`spring.cloud.gcp.core.enabled`	Enables or disables GCP core auto configuration	No	`true`

90.2. Project ID

`GcpProjectIdProvider` is a functional interface that returns a GCP project ID string.

```
public interface GcpProjectIdProvider {
    String getProjectId();
}
```

The Spring Cloud GCP starter auto-configures a `GcpProjectIdProvider`. If a `spring.cloud.gcp.project-id` property is specified, the provided `GcpProjectIdProvider` returns that property value.

```
spring.cloud.gcp.project-id=my-gcp-project-id
```

Otherwise, the project ID is discovered based on an ordered list of rules:

1. The project ID specified by the `GOOGLE_CLOUD_PROJECT` environment variable
2. The Google App Engine project ID
3. The project ID specified in the JSON credentials file pointed by the `GOOGLE_APPLICATION_CREDENTIALS` environment variable
4. The Google Cloud SDK project ID
5. The Google Compute Engine project ID, from the Google Compute Engine Metadata Server

90.3. Credentials

`CredentialsProvider` is a functional interface that returns the credentials to authenticate and authorize calls to Google Cloud Client Libraries.

```
public interface CredentialsProvider {
  Credentials getCredentials() throws IOException;
}
```

The Spring Cloud GCP starter auto-configures a `CredentialsProvider`. It uses the `spring.cloud.gcp.credentials.location` property to locate the OAuth2 private key of a Google service account. Keep in mind this property is a Spring Resource, so the credentials file can be obtained from a number of different locations such as the file system, classpath, URL, etc. The next example specifies the credentials location property in the file system.

```
spring.cloud.gcp.credentials.location=file:/usr/local/key.json
```

Alternatively, you can set the credentials by directly specifying the `spring.cloud.gcp.credentials.encoded-key` property. The value should be the base64-encoded account private key in JSON format.

If that credentials aren't specified through properties, the starter tries to discover credentials from a number of places:

1. Credentials file pointed to by the `GOOGLE_APPLICATION_CREDENTIALS` environment variable
2. Credentials provided by the Google Cloud SDK `gcloud auth application-default login` command
3. Google App Engine built-in credentials
4. Google Cloud Shell built-in credentials
5. Google Compute Engine built-in credentials

If your app is running on Google App Engine or Google Compute Engine, in most cases, you should omit the `spring.cloud.gcp.credentials.location` property and, instead, let the Spring Cloud GCP Starter get the correct credentials for those environments. On App Engine Standard, the App Identity service account credentials are used, on App Engine Flexible, the Flexible service account credential are used and on Google Compute Engine, the Compute Engine Default Service Account is used.

90.3.1. Scopes

By default, the credentials provided by the Spring Cloud GCP Starter contain scopes for every service supported by Spring Cloud GCP.

Service	Scope
Spanner	www.googleapis.com/auth/spanner.admin, www.googleapis.com/auth/spanner.data
Datastore	www.googleapis.com/auth/datastore
Pub/Sub	www.googleapis.com/auth/pubsub
Storage (Read Only)	www.googleapis.com/auth/devstorage.read_only
Storage (Read/Write)	www.googleapis.com/auth/devstorage.read_write
Runtime Config	www.googleapis.com/auth/cloudruntimeconfig
Trace (Append)	www.googleapis.com/auth/trace.append
Cloud Platform	www.googleapis.com/auth/cloud-platform
Vision	www.googleapis.com/auth/cloud-vision

The Spring Cloud GCP starter allows you to configure a custom scope list for the provided credentials. To do that, specify a comma-delimited list of Google OAuth2 scopes in the `spring.cloud.gcp.credentials.scopes` property.

`spring.cloud.gcp.credentials.scopes` is a comma-delimited list of Google OAuth2 scopes for Google Cloud Platform services that the credentials returned by the provided `CredentialsProvider` support.

```
spring.cloud.gcp.credentials.scopes=https://www.googleapis.com/auth/pubsub,https://www
.googleapis.com/auth/sqlservice.admin
```

You can also use `DEFAULT_SCOPES` placeholder as a scope to represent the starters default scopes, and append the additional scopes you need to add.

```
spring.cloud.gcp.credentials.scopes=DEFAULT_SCOPES,https://www.googleapis.com/auth/clo
ud-vision
```

90.4. Environment

`GcpEnvironmentProvider` is a functional interface, auto-configured by the Spring Cloud GCP starter, that returns a `GcpEnvironment` enum. The provider can help determine programmatically in which GCP environment (App Engine Flexible, App Engine Standard, Kubernetes Engine or Compute Engine) the application is deployed.

```
public interface GcpEnvironmentProvider {
    GcpEnvironment getCurrentEnvironment();
}
```

90.5. Spring Initializr

This starter is available from Spring Initializr through the `GCP Support` entry.

Chapter 91. Cloud Storage

Google Cloud Storage allows storing any types of files in single or multiple regions. A Spring Boot starter is provided to auto-configure the various Storage components.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-storage</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
storage'
}
```

This starter is also available from Spring Initializr through the `GCP Storage` entry.

91.1. Using Cloud Storage

The starter automatically configures and registers a `Storage` bean in the Spring application context. The `Storage` bean (Javadoc) can be used to list/create/update/delete buckets (a group of objects with similar permissions and resiliency requirements) and objects.

```
@Autowired
private Storage storage;

public void createFile() {
    Bucket bucket = storage.create(BucketInfo.of("my-app-storage-bucket"));

    storage.create(
        BlobInfo.newBuilder("my-app-storage-bucket", "subdirectory/my-file").build(),
            "file contents".getBytes()
    );
}
```

91.2. Cloud Storage Objects As Spring Resources

Spring Resources are an abstraction for a number of low-level resources, such as file system files, classpath files, servlet context-relative files, etc. Spring Cloud GCP adds a new resource type: a Google Cloud Storage (GCS) object.

The Spring Resource Abstraction for Google Cloud Storage allows GCS objects to be accessed by their GCS URL using the `@Value` annotation:

```
@Value("gs://[YOUR_GCS_BUCKET]/[GCS_FILE_NAME]")
private Resource gcsResource;
```

...or the Spring application context

```
SpringApplication.run(...).getResource("gs://[YOUR_GCS_BUCKET]/[GCS_FILE_NAME]");
```

This creates a `Resource` object that can be used to read the object, among other possible operations.

It is also possible to write to a `Resource`, although a `WriteableResource` is required.

```
@Value("gs://[YOUR_GCS_BUCKET]/[GCS_FILE_NAME]")
private Resource gcsResource;
...
try (OutputStream os = ((WritableResource) gcsResource).getOutputStream()) {
  os.write("foo".getBytes());
}
```

To work with the `Resource` as a Google Cloud Storage resource, cast it to `GoogleStorageResource`.

If the resource path refers to an object on Google Cloud Storage (as opposed to a bucket), then the `getBlob` method can be called to obtain a `Blob`. This type represents a GCS file, which has associated metadata, such as content-type, that can be set. The `createSignedUrl` method can also be used to obtain signed URLs for GCS objects. However, creating signed URLs requires that the resource was created using service account credentials.

The Spring Boot Starter for Google Cloud Storage auto-configures the `Storage` bean required by the `spring-cloud-gcp-storage` module, based on the `CredentialsProvider` provided by the Spring Boot GCP starter.

91.2.1. Setting the Content Type

You can set the content-type of Google Cloud Storage files from their corresponding `Resource` objects:

```
((GoogleStorageResource)gcsResource).getBlob().toBuilder().setContentType("text/html")
.build().update();
```

91.3. Configuration

The Spring Boot Starter for Google Cloud Storage provides the following configuration options:

Name	Description	Required	Default value
`spring.cloud.gcp.storage.enabled`	Enables the GCP storage APIs.	No	`true`
`spring.cloud.gcp.storage.auto-create-files`	Creates files and buckets on Google Cloud Storage when writes are made to non-existent files	No	`true`
`spring.cloud.gcp.storage.credentials.location`	OAuth2 credentials for authenticating with the Google Cloud Storage API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.storage.credentials.encoded-key`	Base64-encoded contents of OAuth2 account private key for authenticating with the Google Cloud Storage API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.storage.credentials.scopes`	OAuth2 scope for Spring Cloud GCP Storage credentials	No	www.googleapis.com/auth/devstorage.read_write

91.4. Sample

A sample application and a codelab are available.

Chapter 92. Cloud SQL

Spring Cloud GCP adds integrations with Spring JDBC so you can run your MySQL or PostgreSQL databases in Google Cloud SQL using Spring JDBC, or other libraries that depend on it like Spring Data JPA.

The Cloud SQL support is provided by Spring Cloud GCP in the form of two Spring Boot starters, one for MySQL and another one for PostgreSQL. The role of the starters is to read configuration from properties and assume default settings so that user experience connecting to MySQL and PostgreSQL is as simple as possible.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-sql-mysql</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-sql-postgresql</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-sql-
mysql'
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-sql-
postgresql'
}
```

92.1. Prerequisites

In order to use the Spring Boot Starters for Google Cloud SQL, the Google Cloud SQL API must be enabled in your GCP project.

To do that, go to the API library page of the Google Cloud Console, search for "Cloud SQL API", click the first result and enable the API.

There are several similar "Cloud SQL" results. You must access the "Google Cloud SQL API" one and enable the API from there.

92.2. Spring Boot Starter for Google Cloud SQL

The Spring Boot Starters for Google Cloud SQL provide an auto-configured `DataSource` object. Coupled with Spring JDBC, it provides a `JdbcTemplate` object bean that allows for operations such as

querying and modifying a database.

```
public List<Map<String, Object>> listUsers() {
    return jdbcTemplate.queryForList("SELECT * FROM user;");
}
```

You can rely on Spring Boot data source auto-configuration to configure a `DataSource` bean. In other words, properties like the SQL username, `spring.datasource.username`, and password, `spring.datasource.password` can be used. There is also some configuration specific to Google Cloud SQL:

Property name	Description	Default value
`spring.cloud.gcp.sql.enabled`	Enables or disables Cloud SQL auto configuration	`true`
`spring.cloud.gcp.sql.database-name`	Name of the database to connect to.	
`spring.cloud.gcp.sql.instance-connection-name`	A string containing a Google Cloud SQL instance's project ID, region and name, each separated by a colon. For example, `my-project-id:my-region:my-instance-name`.	
`spring.cloud.gcp.sql.credentials.location`	File system path to the Google OAuth2 credentials private key file. Used to authenticate and authorize new connections to a Google Cloud SQL instance.	Default credentials provided by the Spring GCP Boot starter
`spring.cloud.gcp.sql.credentials.encoded-key`	Base64-encoded contents of OAuth2 account private key in JSON format. Used to authenticate and authorize new connections to a Google Cloud SQL instance.	Default credentials provided by the Spring GCP Boot starter

If you provide your own `spring.datasource.url`, it will be ignored, unless you disable Cloud SQL auto configuration with `spring.cloud.gcp.sql.enabled=false`.

92.2.1. `DataSource` creation flow

Based on the previous properties, the Spring Boot starter for Google Cloud SQL creates a `CloudSqlJdbcInfoProvider` object which is used to obtain an instance's JDBC URL and driver class name. If you provide your own `CloudSqlJdbcInfoProvider` bean, it is used instead and the properties related to building the JDBC URL or driver class are ignored.

The `DataSourceProperties` object provided by Spring Boot Autoconfigure is mutated in order to use

the JDBC URL and driver class names provided by `CloudSqlJdbcInfoProvider`, unless those values were provided in the properties. It is in the `DataSourceProperties` mutation step that the credentials factory is registered in a system property to be `SqlCredentialFactory`.

`DataSource` creation is delegated to Spring Boot. You can select the type of connection pool (e.g., Tomcat, HikariCP, etc.) by adding their dependency to the classpath.

Using the created `DataSource` in conjunction with Spring JDBC provides you with a fully configured and operational `JdbcTemplate` object that you can use to interact with your SQL database. You can connect to your database with as little as a database and instance names.

92.2.2. Troubleshooting tips

Connection issues

If you're not able to connect to a database and see an endless loop of `Connecting to Cloud SQL instance […] on IP […]`, it's likely that exceptions are being thrown and logged at a level lower than your logger's level. This may be the case with HikariCP, if your logger is set to INFO or higher level.

To see what's going on in the background, you should add a `logback.xml` file to your application resources folder, that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <include resource="org/springframework/boot/logging/logback/base.xml"/>
  <logger name="com.zaxxer.hikari.pool" level="DEBUG"/>
</configuration>
```

Errors like `c.g.cloud.sql.core.SslSocketFactory : Re-throwing cached exception due to attempt to refresh instance information too soon after error`

If you see a lot of errors like this in a loop and can't connect to your database, this is usually a symptom that something isn't right with the permissions of your credentials or the Google Cloud SQL API is not enabled. Verify that the Google Cloud SQL API is enabled in the Cloud Console and that your service account has the necessary IAM roles.

To find out what's causing the issue, you can enable DEBUG logging level as mentioned above.

PostgreSQL: `java.net.SocketException: already connected` issue

We found this exception to be common if your Maven project's parent is `spring-boot` version `1.5.x`, or in any other circumstance that would cause the version of the `org.postgresql:postgresql` dependency to be an older one (e.g., `9.4.1212.jre7`).

To fix this, re-declare the dependency in its correct version. For example, in Maven:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.1.1</version>
</dependency>
```

92.3. Samples

Available sample applications and codelabs:

- Spring Cloud GCP MySQL
- Spring Cloud GCP PostgreSQL
- Spring Data JPA with Spring Cloud GCP SQL
- Codelab: Spring Pet Clinic using Cloud SQL

Chapter 93. Cloud Pub/Sub

Spring Cloud GCP provides an abstraction layer to publish to and subscribe from Google Cloud Pub/Sub topics and to create, list or delete Google Cloud Pub/Sub topics and subscriptions.

A Spring Boot starter is provided to auto-configure the various required Pub/Sub components.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-pubsub</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
pubsub'
}
```

This starter is also available from Spring Initializr through the `GCP Messaging` entry.

93.1. Configuration

The Spring Boot starter for Google Cloud Pub/Sub provides the following configuration options.

93.1.1. Spring Cloud GCP Pub/Sub API Configuration

This section describes options for enabling the integration, specifying the GCP project and credentials, and setting whether the APIs should connect to an emulator for local testing.

Name	Description	Required	Default value
`spring.cloud.gcp.pubsub.enabled`	Enables or disables Pub/Sub auto-configuration	No	`true`
`spring.cloud.gcp.pubsub.project-id`	GCP project ID where the Google Cloud Pub/Sub API is hosted, if different from the one in the Spring Cloud GCP Core Module	No	

`spring.cloud.gcp.pubsub.credentials.location`	OAuth2 credentials for authenticating with the Google Cloud Pub/Sub API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.pubsub.emulator-host`	The host and port of the local running emulator. If provided, this will setup the client to connect against a running Google Cloud Pub/Sub Emulator.	No	
`spring.cloud.gcp.pubsub.credentials.encoded-key`	Base64-encoded contents of OAuth2 account private key for authenticating with the Google Cloud Pub/Sub API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.pubsub.credentials.scopes`	OAuth2 scope for Spring Cloud GCP Pub/Sub credentials	No	www.googleapis.com/auth/pubsub

93.1.2. Publisher/Subscriber Configuration

This section describes configuration options to customize the behavior of the application's Pub/Sub publishers and subscribers.

Name	Description	Required	Default value
`spring.cloud.gcp.pubsub.subscriber.parallel-pull-count`	The number of pull workers	No	1
`spring.cloud.gcp.pubsub.subscriber.max-ack-extension-period`	The maximum period a message ack deadline will be extended, in seconds	No	0
`spring.cloud.gcp.pubsub.subscriber.pull-endpoint`	The endpoint for synchronous pulling messages	No	pubsub.googleapis.com:443
`spring.cloud.gcp.pubsub.[subscriber,publisher].executor-threads`	Number of threads used by `Subscriber` instances created by `SubscriberFactory`	No	4

`spring.cloud.gcp.pubsub.[subscriber,publisher.batching].flow-control.max-outstanding-element-count`	Maximum number of outstanding elements to keep in memory before enforcing flow control.	No	unlimited
`spring.cloud.gcp.pubsub.[subscriber,publisher.batching].flow-control.max-outstanding-request-bytes`	Maximum number of outstanding bytes to keep in memory before enforcing flow control.	No	unlimited
`spring.cloud.gcp.pubsub.[subscriber,publisher.batching].flow-control.limit-exceeded-behavior`	The behavior when the specified limits are exceeded.	No	Block
`spring.cloud.gcp.pubsub.publisher.batching.element-count-threshold`	The element count threshold to use for batching.	No	unset (threshold does not apply)
`spring.cloud.gcp.pubsub.publisher.batching.request-byte-threshold`	The request byte threshold to use for batching.	No	unset (threshold does not apply)
`spring.cloud.gcp.pubsub.publisher.batching.delay-threshold-seconds`	The delay threshold to use for batching. After this amount of time has elapsed (counting from the first element added), the elements will be wrapped up in a batch and sent.	No	unset (threshold does not apply)
`spring.cloud.gcp.pubsub.publisher.batching.enabled`	Enables batching.	No	false

93.1.3. GRPC Connection Settings

The Pub/Sub API uses the GRPC protocol to send API requests to the Pub/Sub service. This section describes configuration options for customizing the GRPC behavior.

The properties that refer to `retry` control the RPC retries for transient failures during the gRPC call to Cloud Pub/Sub server. They do **not** control message redelivery; only message acknowledgement deadline can be used to extend or shorten the amount of time until Pub/Sub attempts redelivery.

Name	Description	Required	Default value

`spring.cloud.gcp.pubsub.keepAliveIntervalMinutes`	Determines frequency of keepalive gRPC ping	No	`5 minutes`
`spring.cloud.gcp.pubsub.[subscriber,publisher].retry.total-timeout-seconds`	TotalTimeout has ultimate control over how long the logic should keep trying the remote call until it gives up completely. The higher the total timeout, the more retries can be attempted.	No	0
`spring.cloud.gcp.pubsub.[subscriber,publisher].retry.initial-retry-delay-second`	InitialRetryDelay controls the delay before the first retry. Subsequent retries will use this value adjusted according to the RetryDelayMultiplier.	No	0
`spring.cloud.gcp.pubsub.[subscriber,publisher].retry.retry-delay-multiplier`	RetryDelayMultiplier controls the change in retry delay. The retry delay of the previous call is multiplied by the RetryDelayMultiplier to calculate the retry delay for the next call.	No	1
`spring.cloud.gcp.pubsub.[subscriber,publisher].retry.max-retry-delay-seconds`	MaxRetryDelay puts a limit on the value of the retry delay, so that the RetryDelayMultiplier can't increase the retry delay higher than this amount.	No	0
`spring.cloud.gcp.pubsub.[subscriber,publisher].retry.max-attempts`	MaxAttempts defines the maximum number of attempts to perform. If this value is greater than 0, and the number of attempts reaches this limit, the logic will give up retrying even if the total retry time is still lower than TotalTimeout.	No	0

`spring.cloud.gcp.pubsub.[subscriber,publisher].retry.jittered`	Jitter determines if the delay time should be randomized.	No	true
`spring.cloud.gcp.pubsub.[subscriber,publisher].retry.initial-rpc-timeout-seconds`	InitialRpcTimeout controls the timeout for the initial RPC. Subsequent calls will use this value adjusted according to the RpcTimeoutMultiplier.	No	0
`spring.cloud.gcp.pubsub.[subscriber,publisher].retry.rpc-timeout-multiplier`	RpcTimeoutMultiplier controls the change in RPC timeout. The timeout of the previous call is multiplied by the RpcTimeoutMultiplier to calculate the timeout for the next call.	No	1
`spring.cloud.gcp.pubsub.[subscriber,publisher].retry.max-rpc-timeout-seconds`	MaxRpcTimeout puts a limit on the value of the RPC timeout, so that the RpcTimeoutMultiplier can't increase the RPC timeout higher than this amount.	No	0

93.2. Spring Boot Actuator Support

93.2.1. Cloud Pub/Sub Health Indicator

If you are using Spring Boot Actuator, you can take advantage of the Cloud Pub/Sub health indicator called `pubsub`. The health indicator will verify whether Cloud Pub/Sub is up and accessible by your application. To enable it, all you need to do is add the Spring Boot Actuator to your project.

The `pubsub` indicator will then roll up to the overall application status visible at localhost:8080/actuator/health (use the `management.endpoint.health.show-details` property to view per-indicator details).

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

If your application already has actuator and Cloud Pub/Sub starters, this health indicator is enabled by default. To disable the Cloud Pub/Sub indicator, set management.health.pubsub.enabled to false.

93.3. Pub/Sub Operations & Template

PubSubOperations is an abstraction that allows Spring users to use Google Cloud Pub/Sub without depending on any Google Cloud Pub/Sub API semantics. It provides the common set of operations needed to interact with Google Cloud Pub/Sub. PubSubTemplate is the default implementation of PubSubOperations and it uses the Google Cloud Java Client for Pub/Sub to interact with Google Cloud Pub/Sub.

93.3.1. Publishing to a topic

PubSubTemplate provides asynchronous methods to publish messages to a Google Cloud Pub/Sub topic. The publish() method takes in a topic name to post the message to, a payload of a generic type and, optionally, a map with the message headers. The topic name could either be a canonical topic name within the current project, or the fully-qualified name referring to a topic in a different project using the projects/<project_name>/topics/<topic_name> format.

Here is an example of how to publish a message to a Google Cloud Pub/Sub topic:

```
Map<String, String> headers = Collections.singletonMap("key1", "val1");
pubSubTemplate.publish(topicName, "message", headers).get();
```

By default, the SimplePubSubMessageConverter is used to convert payloads of type byte[], ByteString, ByteBuffer, and String to Pub/Sub messages.

93.3.2. Subscribing to a subscription

Google Cloud Pub/Sub allows many subscriptions to be associated to the same topic. PubSubTemplate allows you to listen to subscriptions via the subscribe() method. When listening to a subscription, messages will be pulled from Google Cloud Pub/Sub asynchronously and passed to a user provided message handler. The subscription name could either be a canonical subscription name within the current project, or the fully-qualified name referring to a subscription in a different project using the projects/<project_name>/subscriptions/<subscription_name> format.

Example

Subscribe to a subscription with a message handler:

```
Subscriber subscriber = pubSubTemplate.subscribe(subscriptionName, (message) -> {
    logger.info("Message received from " + subscriptionName + " subscription: "
            + message.getPubsubMessage().getData().toStringUtf8());
    message.ack();
});
```

Subscribe methods

PubSubTemplate provides the following subscribe methods:

subscribe(String subscription, Consumer<BasicAcknowledgeablePubsubMessage> messageConsumer)	asynchronously pulls messages and passes them to messageConsumer
subscribeAndConvert(String subscription, Consumer<ConvertedBasicAcknowledgeablePubsubMessage<T>> messageConsumer, Class<T> payloadType)	same as pull, but converts message payload to payloadType using the converter configured in the template

As of version 1.2, subscribing by itself is not enough to keep an application running. For a command-line application, you may want to provide your own ThreadPoolTaskScheduler bean named pubsubSubscriberThreadPool, which by default creates non-daemon threads that will keep an application from stopping. This default behavior has been overridden in Spring Cloud GCP for consistency with Cloud Pub/Sub client library, and to avoid holding up command-line applications that would like to shut down once their work is done.

93.3.3. Pulling messages from a subscription

Google Cloud Pub/Sub supports synchronous pulling of messages from a subscription. This is different from subscribing to a subscription, in the sense that subscribing is an asynchronous task.

Example

Pull up to 10 messages:

```
int maxMessages = 10;
boolean returnImmediately = false;
List<AcknowledgeablePubsubMessage> messages = pubSubTemplate.pull(subscriptionName,
maxMessages,
        returnImmediately);

//acknowledge the messages
pubSubTemplate.ack(messages);

messages.forEach(message ->
logger.info(message.getPubsubMessage().getData().toStringUtf8()));
```

Pull methods

`PubsubTemplate` provides the following pull methods:

pull(String subscription, Integer maxMessages, Boolean returnImmediately)	Pulls a number of messages from a subscription, allowing for the retry settings to be configured. Any messages received by `pull()` are not automatically acknowledged. See Acknowledging messages. The `maxMessages` parameter is the maximum limit of how many messages to pull from a subscription in a single call; this value **must** be greater than 0. You may omit this parameter by passing in `null`; this means there will be no limit on the number of messages pulled (`maxMessages` will be `Integer.MAX_INTEGER`). If `returnImmediately` is `true`, the system will respond immediately even if it there are no messages available to return in the `Pull` response. Otherwise, the system may wait (for a bounded amount of time) until at least one message is available, rather than returning no messages.
pullAndAck	Works the same as the `pull` method and, additionally, acknowledges all received messages.
pullNext	Allows for a single message to be pulled and automatically acknowledged from a subscription.
pullAndConvert	Works the same as the `pull` method and, additionally, converts the Pub/Sub binary payload to an object of the desired type, using the converter configured in the template.

Acknowledging messages

There are two ways to acknowledge messages.

1. To acknowledge multiple messages at once, you can use the `PubSubTemplate.ack()` method. You can also use the `PubSubTemplate.nack()` for negatively acknowledging messages. Using these methods for acknowledging messages in batches is more efficient than acknowledging messages individually, but they **require** the collection of messages to be from the same project.
2. To acknowledge messages individually you can use the `ack()` or `nack()` method on each of them

(to acknowledge or negatively acknowledge, correspondingly).

All `ack()`, `nack()`, and `modifyAckDeadline()` methods on messages, as well as `PubSubSubscriberTemplate`, are implemented asynchronously, returning a `ListenableFuture<Void>` to enable asynchronous processing.

93.3.4. JSON support

For serialization and deserialization of POJOs using Jackson JSON, configure a `PubSubMessageConverter` bean, and the Spring Boot starter for GCP Pub/Sub will automatically wire it into the `PubSubTemplate`.

```
// Note: The ObjectMapper is used to convert Java POJOs to and from JSON.
// You will have to configure your own instance if you are unable to depend
// on the ObjectMapper provided by Spring Boot starters.
@Bean
public PubSubMessageConverter pubSubMessageConverter() {
    return new JacksonPubSubMessageConverter(new ObjectMapper());
}
```

Alternatively, you can set it directly by calling the `setMessageConverter()` method on the `PubSubTemplate`. Other implementations of the `PubSubMessageConverter` can also be configured in the same manner.

Assuming you have the following class defined:

```
static class TestUser {

    String username;

    String password;

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

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

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

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

You can serialize objects to JSON on publish automatically:

```
TestUser user = new TestUser();
user.setUsername("John");
user.setPassword("password");
pubSubTemplate.publish(topicName, user);
```

And that's how you convert messages to objects on pull:

```
int maxMessages = 1;
boolean returnImmediately = false;
List<ConvertedAcknowledgeablePubsubMessage<TestUser>> messages =
pubSubTemplate.pullAndConvert(
        subscriptionName, maxMessages, returnImmediately, TestUser.class);

ConvertedAcknowledgeablePubsubMessage<TestUser> message = messages.get(0);

//acknowledge the message
message.ack();

TestUser receivedTestUser = message.getPayload();
```

Please refer to our Pub/Sub JSON Payload Sample App as a reference for using this functionality.

93.4. Reactive Stream Subscription

It is also possible to acquire a reactive stream backed by a subscription. To do so, a Project Reactor dependency (`io.projectreactor:reactor-core`) must be added to the project. The combination of the Pub/Sub starter and the Project Reactor dependencies will then make a `PubSubReactiveFactory` bean available, which can then be used to get a `Publisher`.

```
@Autowired
PubSubReactiveFactory reactiveFactory;

// ...

Flux<AcknowledgeablePubsubMessage> flux
                = reactiveFactory.poll("exampleSubscription", 1000);
```

The `Flux` then represents an infinite stream of GCP Pub/Sub messages coming in through the specified subscription. For unlimited demand, the Pub/Sub subscription will be polled regularly, at intervals determined by `pollingPeriodMs` parameter passed in when creating the `Flux`. For bounded demand, the `pollingPeriodMs` parameter is unused. Instead, as many messages as possible (up to the requested number) are delivered immediately, with the remaining messages delivered as they become available.

Any exceptions thrown by the underlying message retrieval logic will be passed as an error to the stream. The error handling operators (`Flux#retry()`, `Flux#onErrorResume()` etc.) can be used to recover.

The full range of Project Reactor operations can be applied to the stream. For example, if you only want to fetch 5 messages, you can use `limitRequest` operation to turn the infinite stream into a finite one:

```
Flux<AcknowledgeablePubsubMessage> fiveMessageFlux = flux.limitRequest(5);
```

Messages flowing through the `Flux` should be manually acknowledged.

```
flux.doOnNext(AcknowledgeablePubsubMessage::ack);
```

93.5. Pub/Sub management

`PubSubAdmin` is the abstraction provided by Spring Cloud GCP to manage Google Cloud Pub/Sub resources. It allows for the creation, deletion and listing of topics and subscriptions.

Generally when referring to topics and subscriptions, you can either use the short canonical name within the current project, or the fully-qualified name referring to a topic or subscription in a different project using the `projects/<project_name>/(topics|subscriptions)/<name>` format.

`PubSubAdmin` depends on `GcpProjectIdProvider` and either a `CredentialsProvider` or a `TopicAdminClient` and a `SubscriptionAdminClient`. If given a `CredentialsProvider`, it creates a `TopicAdminClient` and a `SubscriptionAdminClient` with the Google Cloud Java Library for Pub/Sub default settings. The Spring Boot starter for GCP Pub/Sub auto-configures a `PubSubAdmin` object using the `GcpProjectIdProvider` and the `CredentialsProvider` auto-configured by the Spring Boot GCP Core starter.

93.5.1. Creating a topic

`PubSubAdmin` implements a method to create topics:

```
public Topic createTopic(String topicName)
```

Here is an example of how to create a Google Cloud Pub/Sub topic:

```
public void newTopic() {
    pubSubAdmin.createTopic("topicName");
}
```

93.5.2. Deleting a topic

`PubSubAdmin` implements a method to delete topics:

```
public void deleteTopic(String topicName)
```

Here is an example of how to delete a Google Cloud Pub/Sub topic:

```
public void deleteTopic() {
    pubSubAdmin.deleteTopic("topicName");
}
```

93.5.3. Listing topics

`PubSubAdmin` implements a method to list topics:

```
public List<Topic> listTopics
```

Here is an example of how to list every Google Cloud Pub/Sub topic name in a project:

```
List<String> topics = pubSubAdmin
        .listTopics()
        .stream()
        .map(Topic::getName)
        .collect(Collectors.toList());
```

93.5.4. Creating a subscription

`PubSubAdmin` implements a method to create subscriptions to existing topics:

```
public Subscription createSubscription(String subscriptionName, String topicName,
Integer ackDeadline, String pushEndpoint)
```

Here is an example of how to create a Google Cloud Pub/Sub subscription:

```
public void newSubscription() {
    pubSubAdmin.createSubscription("subscriptionName", "topicName", 10,
“https://my.endpoint/push”);
}
```

Alternative methods with default settings are provided for ease of use. The default value for `ackDeadline` is 10 seconds. If `pushEndpoint` isn’t specified, the subscription uses message pulling, instead.

```
public Subscription createSubscription(String subscriptionName, String topicName)
```

```
public Subscription createSubscription(String subscriptionName, String topicName,
Integer ackDeadline)
```

```
public Subscription createSubscription(String subscriptionName, String topicName,
String pushEndpoint)
```

93.5.5. Deleting a subscription

`PubSubAdmin` implements a method to delete subscriptions:

```
public void deleteSubscription(String subscriptionName)
```

Here is an example of how to delete a Google Cloud Pub/Sub subscription:

```
public void deleteSubscription() {
    pubSubAdmin.deleteSubscription("subscriptionName");
}
```

93.5.6. Listing subscriptions

`PubSubAdmin` implements a method to list subscriptions:

```
public List<Subscription> listSubscriptions()
```

Here is an example of how to list every subscription name in a project:

```
List<String> subscriptions = pubSubAdmin
        .listSubscriptions()
        .stream()
        .map(Subscription::getName)
        .collect(Collectors.toList());
```

93.6. Sample

Sample applications for using the template and using a subscription-backed reactive stream are available.

Chapter 94. Spring Integration

Spring Cloud GCP provides Spring Integration adapters that allow your applications to use Enterprise Integration Patterns backed up by Google Cloud Platform services.

94.1. Channel Adapters for Cloud Pub/Sub

The channel adapters for Google Cloud Pub/Sub connect your Spring `MessageChannels` to Google Cloud Pub/Sub topics and subscriptions. This enables messaging between different processes, applications or micro-services backed up by Google Cloud Pub/Sub.

The Spring Integration Channel Adapters for Google Cloud Pub/Sub are included in the `spring-cloud-gcp-pubsub` module and can be autoconfigured by using the `spring-cloud-gcp-starter-pubsub` module in combination with a Spring Integration dependency.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-pubsub</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-core</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
pubsub'
    compile group: 'org.springframework.integration', name: 'spring-integration-core'
}
```

94.1.1. Inbound channel adapter (using Pub/Sub Streaming Pull)

`PubSubInboundChannelAdapter` is the inbound channel adapter for GCP Pub/Sub that listens to a GCP Pub/Sub subscription for new messages. It converts new messages to an internal Spring `Message` and then sends it to the bound output channel.

Google Pub/Sub treats message payloads as byte arrays. So, by default, the inbound channel adapter will construct the Spring `Message` with `byte[]` as the payload. However, you can change the desired payload type by setting the `payloadType` property of the `PubSubInboundChannelAdapter`. The `PubSubInboundChannelAdapter` delegates the conversion to the desired payload type to the `PubSubMessageConverter` configured in the `PubSubTemplate`.

To use the inbound channel adapter, a `PubSubInboundChannelAdapter` must be provided and

configured on the user application side.

```
@Bean
public MessageChannel pubsubInputChannel() {
    return new PublishSubscribeChannel();
}

@Bean
public PubSubInboundChannelAdapter messageChannelAdapter(
    @Qualifier("pubsubInputChannel") MessageChannel inputChannel,
    PubSubSubscriberOperations subscriberOperations) {
    PubSubInboundChannelAdapter adapter =
        new PubSubInboundChannelAdapter(subscriberOperations, "subscriptionName");
    adapter.setOutputChannel(inputChannel);
    adapter.setAckMode(AckMode.MANUAL);

    return adapter;
}
```

In the example, we first specify the `MessageChannel` where the adapter is going to write incoming messages to. The `MessageChannel` implementation isn't important here. Depending on your use case, you might want to use a `MessageChannel` other than `PublishSubscribeChannel`.

Then, we declare a `PubSubInboundChannelAdapter` bean. It requires the channel we just created and a `SubscriberFactory`, which creates `Subscriber` objects from the Google Cloud Java Client for Pub/Sub. The Spring Boot starter for GCP Pub/Sub provides a configured `PubSubSubscriberOperations` object.

Acknowledging messages and handling failures

When working with Cloud Pub/Sub, it is important to understand the concept of `ackDeadline` — the amount of time Cloud Pub/Sub will wait until attempting redelivery of an outstanding message. Each subscription has a default `ackDeadline` applied to all messages sent to it. Additionally, the Cloud Pub/Sub client library can extend each streamed message's `ackDeadline` until the message processing completes, fails or until the maximum extension period elapses.

In the Pub/Sub client library, default maximum extension period is an hour. However, Spring Cloud GCP disables this auto-extension behavior. Use the `spring.cloud.gcp.pubsub.subscriber.max-ack-extension-period` property to re-enable it.

Acknowledging (acking) a message removes it from Pub/Sub's known outstanding messages. Nacking a message resets its acknowledgement deadline to 0, forcing immediate redelivery. This could be useful in a load balanced architecture, where one of the subscribers is having issues but others are available to process messages.

The `PubSubInboundChannelAdapter` supports three acknowledgement modes: the default `AckMode.AUTO` (automatic acking on processing success and nacking on exception), as well as two modes for additional manual control: AckMode.AUTO_ACK (automatic acking on success but no action on exception) and AckMode.MANUAL (no automatic actions at all; both acking and nacking have to be

done manually).

Table 5. Acknowledgement mode behavior

	AUTO	AUTO_ACK	MANUAL
Message processing completes successfully	ack, no redelivery	ack, no redelivery	<no action>*
Message processing fails, but error handler completes successfully**	ack, no redelivery	ack, no redelivery	<no action>*
Message processing fails; no error handler present	nack, immediate redelivery	<no action>*	<no action>*
Message processing fails, and error handler throws an exception	nack, immediate redelivery	<no action>*	<no action>*

* <no action> means that the message will be neither acked nor nacked. Cloud Pub/Sub will attempt redelivery according to subscription `ackDeadline` setting and the `max-ack-extension-period` client library setting.

** For the adapter, "success" means the Spring Integration flow processed without raising an exception, so successful message processing and the successful completion of an error handler both result in the same behavior (message will be acknowledged). To trigger default error behavior (nacking in `AUTO` mode; neither acking nor nacking in `AUTO_ACK` mode), propagate the error back to the adapter by throwing an exception from the Error Handling flow.

Manual acking/nacking

The adapter attaches a `BasicAcknowledgeablePubsubMessage` object to the `Message` headers. Users can extract the `BasicAcknowledgeablePubsubMessage` using the `GcpPubSubHeaders.ORIGINAL_MESSAGE` key and use it to ack (or nack) a message.

```
@Bean
@ServiceActivator(inputChannel = "pubsubInputChannel")
public MessageHandler messageReceiver() {
    return message -> {
        LOGGER.info("Message arrived! Payload: " + new String((byte[])
message.getPayload()));
        BasicAcknowledgeablePubsubMessage originalMessage =
              message.getHeaders().get(GcpPubSubHeaders.ORIGINAL_MESSAGE,
BasicAcknowledgeablePubsubMessage.class);
        originalMessage.ack();
    };
}
```

Error Handling

If you want to have more control over message processing in case of an error, you need to associate the `PubSubInboundChannelAdapter` with a Spring Integration error channel and specify the behavior to be invoked with `@ServiceActivator`.

In order to activate the default behavior (nacking in `AUTO` mode; neither acking nor nacking in `AUTO_ACK` mode), your error handler has to throw an exception. Otherwise, the adapter will assume that processing completed successfully and will ack the message.

```
@Bean
public MessageChannel pubsubInputChannel() {
    return new PublishSubscribeChannel();
}

@Bean
public PubSubInboundChannelAdapter messageChannelAdapter(
    @Qualifier("pubsubInputChannel") MessageChannel inputChannel,
    SubscriberFactory subscriberFactory) {
    PubSubInboundChannelAdapter adapter =
        new PubSubInboundChannelAdapter(subscriberFactory, "subscriptionName");
    adapter.setOutputChannel(inputChannel);
    adapter.setAckMode(AckMode.AUTO_ACK);
    adapter.setErrorChannelName("pubsubErrors");

    return adapter;
}

@ServiceActivator(inputChannel =  "pubsubErrors")
public void pubsubErrorHandler(Message<MessagingException> message) {
    LOGGER.warn("This message will be automatically acked because error handler
completes successfully");
}
```

If you would prefer to manually ack or nack the message, you can do it by retrieving the header of the exception payload:

```
@ServiceActivator(inputChannel =  "pubsubErrors")
public void pubsubErrorHandler(Message<MessagingException> exceptionMessage) {

    BasicAcknowledgeablePubsubMessage originalMessage =

(BasicAcknowledgeablePubsubMessage)exceptionMessage.getPayload().getFailedMessage()
        .getHeaders().get(GcpPubSubHeaders.ORIGINAL_MESSAGE);

    originalMessage.nack();
}
```

94.1.2. Pollable Message Source (using Pub/Sub Synchronous Pull)

While `PubSubInboundChannelAdapter`, through the underlying Asynchronous Pull Pub/Sub mechanism, provides the best performance for high-volume applications that receive a steady flow of messages, it can create load balancing anomalies due to message caching. This behavior is most obvious when publishing a large batch of small messages that take a long time to process individually. It manifests as one subscriber taking up most messages, even if multiple subscribers are available to take on the work. For a more detailed explanation of this scenario, see GCP Pub/Sub documentation.

In such a scenario, a `PubSubMessageSource` can help spread the load between different subscribers more evenly.

As with the Inbound Channel Adapter, the message source has a configurable acknowledgement mode, payload type, and header mapping.

The default behavior is to return from the synchronous pull operation immediately if no messages are present. This can be overridden by using `setBlockOnPull()` method to wait for at least one message to arrive.

By default, `PubSubMessageSource` pulls from the subscription one message at a time. To pull a batch of messages on each request, use the `setMaxFetchSize()` method to set the batch size.

```
@Bean
@InboundChannelAdapter(channel = "pubsubInputChannel", poller = @Poller(fixedDelay =
"100"))
public MessageSource<Object> pubsubAdapter(PubSubTemplate pubSubTemplate) {
    PubSubMessageSource messageSource = new PubSubMessageSource(pubSubTemplate,
"exampleSubscription");
    messageSource.setAckMode(AckMode.MANUAL);
    messageSource.setPayloadType(String.class);
    messageSource.setBlockOnPull(true);
    messageSource.setMaxFetchSize(100);
    return messageSource;
}
```

The `@InboundChannelAdapter` annotation above ensures that the configured `MessageSource` is polled for messages, which are then available for manipulation with any Spring Integration mechanism on the `pubsubInputChannel` message channel. For example, messages can be retrieved in a method annotated with `@ServiceActivator`, as seen below.

For additional flexibility, `PubSubMessageSource` attaches an `AcknowledgeablePubSubMessage` object to the `GcpPubSubHeaders.ORIGINAL_MESSAGE` message header. The object can be used for manually (n)acking the message.

```
@ServiceActivator(inputChannel = "pubsubInputChannel")
public void messageReceiver(String payload,
        @Header(GcpPubSubHeaders.ORIGINAL_MESSAGE) AcknowledgeablePubsubMessage
message)
            throws InterruptedException {
    LOGGER.info("Message arrived by Synchronous Pull! Payload: " + payload);
    message.ack();
}
```

AcknowledgeablePubSubMessage objects acquired by synchronous pull are aware of their own acknowledgement IDs. Streaming pull does not expose this information due to limitations of the underlying API, and returns BasicAcknowledgeablePubsubMessage objects that allow acking/nacking individual messages, but not extracting acknowledgement IDs for future processing.

94.1.3. Outbound channel adapter

PubSubMessageHandler is the outbound channel adapter for GCP Pub/Sub that listens for new messages on a Spring MessageChannel. It uses PubSubTemplate to post them to a GCP Pub/Sub topic.

To construct a Pub/Sub representation of the message, the outbound channel adapter needs to convert the Spring Message payload to a byte array representation expected by Pub/Sub. It delegates this conversion to the PubSubTemplate. To customize the conversion, you can specify a PubSubMessageConverter in the PubSubTemplate that should convert the Object payload and headers of the Spring Message to a PubsubMessage.

To use the outbound channel adapter, a PubSubMessageHandler bean must be provided and configured on the user application side.

```
@Bean
@ServiceActivator(inputChannel = "pubsubOutputChannel")
public MessageHandler messageSender(PubSubTemplate pubsubTemplate) {
    return new PubSubMessageHandler(pubsubTemplate, "topicName");
}
```

The provided PubSubTemplate contains all the necessary configuration to publish messages to a GCP Pub/Sub topic.

PubSubMessageHandler publishes messages asynchronously by default. A publish timeout can be configured for synchronous publishing. If none is provided, the adapter waits indefinitely for a response.

It is possible to set user-defined callbacks for the publish() call in PubSubMessageHandler through the setPublishFutureCallback() method. These are useful to process the message ID, in case of success, or the error if any was thrown.

To override the default destination you can use the GcpPubSubHeaders.DESTINATION header.

```
@Autowired
private MessageChannel pubsubOutputChannel;

public void handleMessage(Message<?> msg) throws MessagingException {
    final Message<?> message = MessageBuilder
        .withPayload(msg.getPayload())
        .setHeader(GcpPubSubHeaders.TOPIC, "customTopic").build();
    pubsubOutputChannel.send(message);
}
```

It is also possible to set an SpEL expression for the topic with the `setTopicExpression()` or `setTopicExpressionString()` methods.

94.1.4. Header mapping

These channel adapters contain header mappers that allow you to map, or filter out, headers from Spring to Google Cloud Pub/Sub messages, and vice-versa. By default, the inbound channel adapter maps every header on the Google Cloud Pub/Sub messages to the Spring messages produced by the adapter. The outbound channel adapter maps every header from Spring messages into Google Cloud Pub/Sub ones, except the ones added by Spring, like headers with key `"id"`, `"timestamp"` and `"gcp_pubsub_acknowledgement"`. In the process, the outbound mapper also converts the value of the headers into string.

Each adapter declares a `setHeaderMapper()` method to let you further customize which headers you want to map from Spring to Google Cloud Pub/Sub, and vice-versa.

For example, to filter out headers `"foo"`, `"bar"` and all headers starting with the prefix "prefix_", you can use `setHeaderMapper()` along with the `PubSubHeaderMapper` implementation provided by this module.

```
PubSubMessageHandler adapter = ...
...
PubSubHeaderMapper headerMapper = new PubSubHeaderMapper();
headerMapper.setOutboundHeaderPatterns("!foo", "!bar", "!prefix_*", "*");
adapter.setHeaderMapper(headerMapper);
```

The order in which the patterns are declared in `PubSubHeaderMapper.setOutboundHeaderPatterns()` and `PubSubHeaderMapper.setInboundHeaderPatterns()` matters. The first patterns have precedence over the following ones.

In the previous example, the `"*"` pattern means every header is mapped. However, because it comes last in the list, the previous patterns take precedence.

94.1.5. Samples

Available examples:

- Sending/Receiving Messages with Channel Adapters
- Pub/Sub Channel Adapters with JSON payloads
- Spring Integration and Pub/Sub Codelab

94.2. Channel Adapters for Google Cloud Storage

The channel adapters for Google Cloud Storage allow you to read and write files to Google Cloud Storage through `MessageChannels`.

Spring Cloud GCP provides two inbound adapters, `GcsInboundFileSynchronizingMessageSource` and `GcsStreamingMessageSource`, and one outbound adapter, `GcsMessageHandler`.

The Spring Integration Channel Adapters for Google Cloud Storage are included in the `spring-cloud-gcp-storage` module.

To use the Storage portion of Spring Integration for Spring Cloud GCP, you must also provide the `spring-integration-file` dependency, since it isn't pulled transitively.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-storage</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-file</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
storage'
    compile group: 'org.springframework.integration', name: 'spring-integration-file'
}
```

94.2.1. Inbound channel adapter

The Google Cloud Storage inbound channel adapter polls a Google Cloud Storage bucket for new files and sends each of them in a `Message` payload to the `MessageChannel` specified in the `@InboundChannelAdapter` annotation. The files are temporarily stored in a folder in the local file system.

Here is an example of how to configure a Google Cloud Storage inbound channel adapter.

```
@Bean
@InboundChannelAdapter(channel = "new-file-channel", poller = @Poller(fixedDelay =
"5000"))
public MessageSource<File> synchronizerAdapter(Storage gcs) {
  GcsInboundFileSynchronizer synchronizer = new GcsInboundFileSynchronizer(gcs);
  synchronizer.setRemoteDirectory("your-gcs-bucket");

  GcsInboundFileSynchronizingMessageSource synchAdapter =
          new GcsInboundFileSynchronizingMessageSource(synchronizer);
  synchAdapter.setLocalDirectory(new File("local-directory"));

  return synchAdapter;
}
```

94.2.2. Inbound streaming channel adapter

The inbound streaming channel adapter is similar to the normal inbound channel adapter, except it does not require files to be stored in the file system.

Here is an example of how to configure a Google Cloud Storage inbound streaming channel adapter.

```
@Bean
@InboundChannelAdapter(channel = "streaming-channel", poller = @Poller(fixedDelay =
"5000"))
public MessageSource<InputStream> streamingAdapter(Storage gcs) {
  GcsStreamingMessageSource adapter =
          new GcsStreamingMessageSource(new GcsRemoteFileTemplate(new
GcsSessionFactory(gcs)));
  adapter.setRemoteDirectory("your-gcs-bucket");
  return adapter;
}
```

If you would like to process the files in your bucket in a specific order, you may pass in a `Comparator<BlobInfo>` to the constructor `GcsStreamingMessageSource` to sort the files being processed.

94.2.3. Outbound channel adapter

The outbound channel adapter allows files to be written to Google Cloud Storage. When it receives a `Message` containing a payload of type `File`, it writes that file to the Google Cloud Storage bucket specified in the adapter.

Here is an example of how to configure a Google Cloud Storage outbound channel adapter.

```
@Bean
@ServiceActivator(inputChannel = "writeFiles")
public MessageHandler outboundChannelAdapter(Storage gcs) {
  GcsMessageHandler outboundChannelAdapter = new GcsMessageHandler(new
GcsSessionFactory(gcs));
  outboundChannelAdapter.setRemoteDirectoryExpression(new ValueExpression<>("your-gcs-
bucket"));

  return outboundChannelAdapter;
}
```

94.2.4. Sample

See the Spring Integration with Google Cloud Storage Sample Code.

Chapter 95. Spring Cloud Stream

Spring Cloud GCP provides a Spring Cloud Stream binder to Google Cloud Pub/Sub.

The provided binder relies on the Spring Integration Channel Adapters for Google Cloud Pub/Sub.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-pubsub-stream-binder</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-pubsub-stream-
binder'
}
```

95.1. Overview

This binder binds producers to Google Cloud Pub/Sub topics and consumers to subscriptions.

Partitioning is currently not supported by this binder.

95.2. Configuration

You can configure the Spring Cloud Stream Binder for Google Cloud Pub/Sub to automatically generate the underlying resources, like the Google Cloud Pub/Sub topics and subscriptions for producers and consumers. For that, you can use the `spring.cloud.stream.gcp.pubsub.bindings.<channelName>.<consumer|producer>.auto-create-resources` property, which is turned ON by default.

Starting with version 1.1, these and other binder properties can be configured globally for all the bindings, e.g. `spring.cloud.stream.gcp.pubsub.default.consumer.auto-create-resources`.

If you are using Pub/Sub auto-configuration from the Spring Cloud GCP Pub/Sub Starter, you should refer to the configuration section for other Pub/Sub parameters.

To use this binder with a running emulator, configure its host and port via `spring.cloud.gcp.pubsub.emulator-host`.

95.2.1. Producer Synchronous Sending Configuration

By default, this binder will send messages to Cloud Pub/Sub asynchronously. If synchronous sending is preferred (for example, to allow propagating errors back to the sender), set `spring.cloud.stream.gcp.pubsub.default.producer.sync` property to `true`.

95.2.2. Producer Destination Configuration

If automatic resource creation is turned ON and the topic corresponding to the destination name does not exist, it will be created.

For example, for the following configuration, a topic called `myEvents` would be created.

application.properties

```
spring.cloud.stream.bindings.events.destination=myEvents
spring.cloud.stream.gcp.pubsub.bindings.events.producer.auto-create-resources=true
```

95.2.3. Consumer Destination Configuration

A `PubSubInboundChannelAdapter` will be configured for your consumer endpoint. You may adjust the ack mode of the consumer endpoint using the `ack-mode` property. The ack mode controls how messages will be acknowledged when they are successfully received. The three possible options are: `AUTO` (default), `AUTO_ACK`, and `MANUAL`. These options are described in detail in the Pub/Sub channel adapter documentation.

application.properties

```
# How to set the ACK mode of the consumer endpoint.
spring.cloud.stream.gcp.pubsub.bindings.{CONSUMER_NAME}.consumer.ack-mode=AUTO_ACK
```

If automatic resource creation is turned ON and the subscription and/or the topic do not exist for a consumer, a subscription and potentially a topic will be created. The topic name will be the same as the destination name, and the subscription name will be the destination name followed by the consumer group name.

Regardless of the `auto-create-resources` setting, if the consumer group is not specified, an anonymous one will be created with the name `anonymous.<destinationName>.<randomUUID>`. Then when the binder shuts down, all Pub/Sub subscriptions created for anonymous consumer groups will be automatically cleaned up.

For example, for the following configuration, a topic named `myEvents` and a subscription called `myEvents.consumerGroup1` would be created. If the consumer group is not specified, a subscription called `anonymous.myEvents.a6d83782-c5a3-4861-ac38-e6e2af15a7be` would be created and later cleaned up.

If you are manually creating Pub/Sub subscriptions for consumers, make sure that they follow the naming convention of `<destinationName>.<consumerGroup>`.

application.properties

```
spring.cloud.stream.bindings.events.destination=myEvents
spring.cloud.stream.gcp.pubsub.bindings.events.consumer.auto-create-resources=true

# specify consumer group, and avoid anonymous consumer group generation
spring.cloud.stream.bindings.events.group=consumerGroup1
```

95.3. Binding with Functions

Since version 3.0, Spring Cloud Stream supports a functional programming model natively. This means that the only requirement for turning your application into a sink is presence of a `java.util.function.Consumer` bean in the application context.

```
@Bean
public Consumer<UserMessage> logUserMessage() {
  return userMessage -> {
    // process message
  }
};
```

A source application is one where a `Supplier` bean is present. It can return an object, in which case Spring Cloud Stream will invoke the supplier repeatedly. Alternatively, the function can return a reactive stream, which will be used as is.

```
@Bean
Supplier<Flux<UserMessage>> generateUserMessages() {
  return () -> /* flux creation logic */;
}
```

A processor application works similarly to a source application, except it is triggered by presence of a `Function` bean.

95.4. Binding with Annotations

As of version 3.0, annotation binding is considered legacy.

To set up a sink application in this style, you would associate a class with a binding interface, such as the built-in `Sink` interface.

```
@EnableBinding(Sink.class)
public class SinkExample {

    @StreamListener(Sink.INPUT)
    public void handleMessage(UserMessage userMessage) {
        // process message
    }
}
```

To set up a source application, you would similarly associate a class with a built-in `Source` interface, and inject an instance of it provided by Spring Cloud Stream.

```
@EnableBinding(Source.class)
public class SourceExample {

    @Autowired
    private Source source;

    public void sendMessage() {
        this.source.output().send(new GenericMessage<>(/* your object here */));
    }
}
```

95.5. Streaming vs. Polled Input

Many Spring Cloud Stream applications will use the built-in `Sink` binding, which triggers the *streaming* input binder creation. Messages can then be consumed with an input handler marked by `@StreamListener(Sink.INPUT)` annotation, at whatever rate Pub/Sub sends them.

For more control over the rate of message arrival, a polled input binder can be set up by defining a custom binding interface with an `@Input`-annotated method returning `PollableMessageSource`.

```
public interface PollableSink {

    @Input("input")
    PollableMessageSource input();
}
```

The `PollableMessageSource` can then be injected and queried, as needed.

```
@EnableBinding(PollableSink.class)
public class SinkExample {

    @Autowired
    PollableMessageSource destIn;

    @Bean
    public ApplicationRunner singlePollRunner() {
        return args -> {
            // This will poll only once.
            // Add a loop or a scheduler to get more messages.
            destIn.poll((message) -> System.out.println("Message retrieved: " +
message));
        };
    }
}
```

95.6. Sample

Sample applications are available:

- For streaming input, annotation-based.
- For streaming input, functional style.
- For polled input.

Chapter 96. Spring Cloud Bus

Using Cloud Pub/Sub as the Spring Cloud Bus implementation is as simple as importing the spring-cloud-gcp-starter-bus-pubsub starter.

This starter brings in the Spring Cloud Stream binder for Cloud Pub/Sub, which is used to both publish and subscribe to the bus. If the bus topic (named springCloudBus by default) does not exist, the binder automatically creates it. The binder also creates anonymous subscriptions for each project using the spring-cloud-gcp-starter-bus-pubsub starter.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-bus-pubsub</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-bus-
pubsub'
}
```

96.1. Configuration Management with Spring Cloud Config and Spring Cloud Bus

Spring Cloud Bus can be used to push configuration changes from a Spring Cloud Config server to the clients listening on the same bus.

To use GCP Pub/Sub as the bus implementation, both the configuration server and the configuration client need the spring-cloud-gcp-starter-bus-pubsub dependency.

All other configuration is standard to Spring Cloud Config.

[spring cloud bus over pubsub] | *spring_cloud_bus_over_pubsub.png*

Spring Cloud Config Server typically runs on port 8888, and can read configuration from a variety of source control systems such as GitHub, and even from the local filesystem. When the server is notified that new configuration is available, it fetches the updated configuration and sends a notification (RefreshRemoteApplicationEvent) out via Spring Cloud Bus.

When configuration is stored locally, config server polls the parent directory for changes. With configuration stored in source control repository, such as GitHub, the config server needs to be notified that a new version of configuration is available. In a deployed server, this would be done automatically through a GitHub webhook, but in a local testing scenario, the /monitor HTTP

endpoint needs to be invoked manually.

```
curl -X POST http://localhost:8888/monitor -H "X-Github-Event: push" -H "Content-Type:
application/json" -d '{"commits": [{"modified": ["application.properties"]}]}'
```

By adding the `spring-cloud-gcp-starter-bus-pubsub` dependency, you instruct Spring Cloud Bus to use Cloud Pub/Sub to broadcast configuration changes. Spring Cloud Bus will then create a topic named `springCloudBus`, as well as a subscription for each configuration client.

The configuration server happens to also be a configuration client, subscribing to the configuration changes that it sends out. Thus, in a scenario with one configuration server and one configuration client, two anonymous subscriptions to the `springCloudBus` topic are created. However, a config server disables configuration refresh by default (see ConfigServerBootstrapApplicationListener for more details).

A demo application showing configuration management and distribution over a Cloud Pub/Sub-powered bus is available. The sample contains two examples of configuration management with Spring Cloud Bus: one monitoring a local file system, and the other retrieving configuration from a GitHub repository.

Chapter 97. Stackdriver Trace

Google Cloud Platform provides a managed distributed tracing service called Stackdriver Trace, and Spring Cloud Sleuth can be used with it to easily instrument Spring Boot applications for observability.

Typically, Spring Cloud Sleuth captures trace information and forwards traces to services like Zipkin for storage and analysis. However, on GCP, instead of running and maintaining your own Zipkin instance and storage, you can use Stackdriver Trace to store traces, view trace details, generate latency distributions graphs, and generate performance regression reports.

This Spring Cloud GCP starter can forward Spring Cloud Sleuth traces to Stackdriver Trace without an intermediary Zipkin server.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-trace</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-trace'
}
```

You must enable Stackdriver Trace API from the Google Cloud Console in order to capture traces. Navigate to the Stackdriver Trace API for your project and make sure it's enabled.

If you are already using a Zipkin server capturing trace information from multiple platform/frameworks, you can also use a Stackdriver Zipkin proxy to forward those traces to Stackdriver Trace without modifying existing applications.

97.1. Tracing

Spring Cloud Sleuth uses the Brave tracer to generate traces. This integration enables Brave to use the `StackdriverTracePropagation` propagation.

A propagation is responsible for extracting trace context from an entity (e.g., an HTTP servlet request) and injecting trace context into an entity. A canonical example of the propagation usage is a web server that receives an HTTP request, which triggers other HTTP requests from the server before returning an HTTP response to the original caller. In the case of `StackdriverTracePropagation`, first it looks for trace context in the `x-cloud-trace-context` key (e.g., an HTTP request header). The value of the `x-cloud-trace-context` key can be formatted in three different ways:

- `x-cloud-trace-context: TRACE_ID`
- `x-cloud-trace-context: TRACE_ID/SPAN_ID`
- `x-cloud-trace-context: TRACE_ID/SPAN_ID;o=TRACE_TRUE`

`TRACE_ID` is a 32-character hexadecimal value that encodes a 128-bit number.

`SPAN_ID` is an unsigned long. Since Stackdriver Trace doesn't support span joins, a new span ID is always generated, regardless of the one specified in `x-cloud-trace-context`.

`TRACE_TRUE` can either be `0` if the entity should be untraced, or `1` if it should be traced. This field forces the decision of whether or not to trace the request; if omitted then the decision is deferred to the sampler.

If a `x-cloud-trace-context` key isn't found, `StackdriverTracePropagation` falls back to tracing with the X-B3 headers.

97.2. Spring Boot Starter for Stackdriver Trace

Spring Boot Starter for Stackdriver Trace uses Spring Cloud Sleuth and auto-configures a StackdriverSender that sends the Sleuth's trace information to Stackdriver Trace.

All configurations are optional:

Name	Description	Required	Default value
`spring.cloud.gcp.trace.enabled`	Auto-configure Spring Cloud Sleuth to send traces to Stackdriver Trace.	No	`true`
`spring.cloud.gcp.trace.project-id`	Overrides the project ID from the Spring Cloud GCP Module	No	
`spring.cloud.gcp.trace.credentials.location`	Overrides the credentials location from the Spring Cloud GCP Module	No	
`spring.cloud.gcp.trace.credentials.encoded-key`	Overrides the credentials encoded key from the Spring Cloud GCP Module	No	
`spring.cloud.gcp.trace.credentials.scopes`	Overrides the credentials scopes from the Spring Cloud GCP Module	No	
`spring.cloud.gcp.trace.num-executor-threads`	Number of threads used by the Trace executor	No	4

`spring.cloud.gcp.trace.authority`	HTTP/2 authority the channel claims to be connecting to.	No	
`spring.cloud.gcp.trace.compression`	Name of the compression to use in Trace calls	No	
`spring.cloud.gcp.trace.deadline-ms`	Call deadline in milliseconds	No	
`spring.cloud.gcp.trace.max-inbound-size`	Maximum size for inbound messages	No	
`spring.cloud.gcp.trace.max-outbound-size`	Maximum size for outbound messages	No	
`spring.cloud.gcp.trace.wait-for-ready`	Waits for the channel to be ready in case of a transient failure	No	`false`
`spring.cloud.gcp.trace.messageTimeout`	Timeout in seconds before pending spans will be sent in batches to GCP Stackdriver Trace. (previously `spring.zipkin.messageTimeout`)	No	1

You can use core Spring Cloud Sleuth properties to control Sleuth's sampling rate, etc. Read Sleuth documentation for more information on Sleuth configurations.

For example, when you are testing to see the traces are going through, you can set the sampling rate to 100%.

```
spring.sleuth.sampler.probability=1                     # Send 100% of the request
traces to Stackdriver.
spring.sleuth.web.skipPattern=(^cleanup.*|.+favicon.*)  # Ignore some URL paths.
spring.sleuth.scheduled.enabled=false                   # disable executor 'async'
traces
```

By default, Spring Cloud Sleuth auto-configuration instruments executor beans, which may cause recurring traces with the name `async` to appear in Stackdriver Trace if your application or one of its dependencies introduces scheduler beans into Spring application context. To avoid this noise, please disable automatic instrumentation of executors via `spring.sleuth.scheduled.enabled=false` in your application configuration.

Spring Cloud GCP Trace does override some Sleuth configurations:

- Always uses 128-bit Trace IDs. This is required by Stackdriver Trace.

- Does not use Span joins. Span joins will share the span ID between the client and server Spans. Stackdriver requires that every Span ID within a Trace to be unique, so Span joins are not supported.
- Uses `StackdriverHttpClientParser` and `StackdriverHttpServerParser` by default to populate Stackdriver related fields.

97.3. Overriding the auto-configuration

Spring Cloud Sleuth supports sending traces to multiple tracing systems as of version 2.1.0. In order to get this to work, every tracing system needs to have a `Reporter<Span>` and `Sender`. If you want to override the provided beans you need to give them a specific name. To do this you can use respectively `StackdriverTraceAutoConfiguration.REPORTER_BEAN_NAME` and `StackdriverTraceAutoConfiguration.SENDER_BEAN_NAME`.

97.4. Customizing spans

You can add additional tags and annotations to spans by using the `brave.SpanCustomizer`, which is available in the application context.

Here's an example that uses `WebMvcConfigurer` to configure an MVC interceptor that adds two extra tags to all web controller spans.

```
@SpringBootApplication
public class Application implements WebMvcConfigurer {

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

    @Autowired
    private SpanCustomizer spanCustomizer;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new HandlerInterceptor() {
            @Override
            public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
                spanCustomizer.tag("session-id", request.getSession().getId());
                spanCustomizer.tag("environment", "QA");

                return true;
            }
        });
    }
}
```

You can then search and filter traces based on these additional tags in the Stackdriver Trace service.

97.5. Integration with Logging

Integration with Stackdriver Logging is available through the Stackdriver Logging Support. If the Trace integration is used together with the Logging one, the request logs will be associated to the corresponding traces. The trace logs can be viewed by going to the Google Cloud Console Trace List, selecting a trace and pressing the `Logs → View` link in the `Details` section.

97.6. Sample

A sample application and a codelab are available.

Chapter 98. Stackdriver Logging

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-logging</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
logging'
}
```

Stackdriver Logging is the managed logging service provided by Google Cloud Platform.

This module provides support for associating a web request trace ID with the corresponding log entries. It does so by retrieving the `X-B3-TraceId` value from the Mapped Diagnostic Context (MDC), which is set by Spring Cloud Sleuth. If Spring Cloud Sleuth isn't used, the configured `TraceIdExtractor` extracts the desired header value and sets it as the log entry's trace ID. This allows grouping of log messages by request, for example, in the Google Cloud Console Logs viewer.

Due to the way logging is set up, the GCP project ID and credentials defined in `application.properties` are ignored. Instead, you should set the `GOOGLE_CLOUD_PROJECT` and `GOOGLE_APPLICATION_CREDENTIALS` environment variables to the project ID and credentials private key location, respectively. You can do this easily if you're using the Google Cloud SDK, using the `gcloud config set project [YOUR_PROJECT_ID]` and `gcloud auth application-default login` commands, respectively.

98.1. Web MVC Interceptor

For use in Web MVC-based applications, `TraceIdLoggingWebMvcInterceptor` is provided that extracts the request trace ID from an HTTP request using a `TraceIdExtractor` and stores it in a thread-local, which can then be used in a logging appender to add the trace ID metadata to log messages.

If Spring Cloud GCP Trace is enabled, the logging module disables itself and delegates log correlation to Spring Cloud Sleuth.

`LoggingWebMvcConfigurer` configuration class is also provided to help register the `TraceIdLoggingWebMvcInterceptor` in Spring MVC applications.

Applications hosted on the Google Cloud Platform include trace IDs under the `x-cloud-trace-context` header, which will be included in log entries. However, if Sleuth is used the trace ID will be

picked up from the MDC.

98.2. Logback Support

Currently, only Logback is supported and there are 2 possibilities to log to Stackdriver via this library with Logback: via direct API calls and through JSON-formatted console logs.

98.2.1. Log via API

A Stackdriver appender is available using `org/springframework/cloud/gcp/logging/logback-appender.xml`. This appender builds a Stackdriver Logging log entry from a JUL or Logback log entry, adds a trace ID to it and sends it to Stackdriver Logging.

`STACKDRIVER_LOG_NAME` and `STACKDRIVER_LOG_FLUSH_LEVEL` environment variables can be used to customize the `STACKDRIVER` appender.

Your configuration may then look like this:

```
<configuration>
  <include resource="org/springframework/cloud/gcp/logging/logback-appender.xml" />

  <root level="INFO">
    <appender-ref ref="STACKDRIVER" />
  </root>
</configuration>
```

If you want to have more control over the log output, you can further configure the appender. The following properties are available:

Property	Default Value	Description
`log`	`spring.log`	The Stackdriver Log name. This can also be set via the `STACKDRIVER_LOG_NAME` environmental variable.
`flushLevel`	`WARN`	If a log entry with this level is encountered, trigger a flush of locally buffered log to Stackdriver Logging. This can also be set via the `STACKDRIVER_LOG_FLUSH_LEVEL` environmental variable.

98.2.2. Log via Console

For Logback, a `org/springframework/cloud/gcp/logging/logback-json-appender.xml` file is made available for import to make it easier to configure the JSON Logback appender.

Your configuration may then look something like this:

```
<configuration>
  <include resource="org/springframework/cloud/gcp/logging/logback-json-appender.xml"
/>

  <root level="INFO">
    <appender-ref ref="CONSOLE_JSON" />
  </root>
</configuration>
```

If your application is running on Google Kubernetes Engine, Google Compute Engine or Google App Engine Flexible, your console logging is automatically saved to Google Stackdriver Logging. Therefore, you can just include `org/springframework/cloud/gcp/logging/logback-json-appender.xml` in your logging configuration, which logs JSON entries to the console. The trace id will be set correctly.

If you want to have more control over the log output, you can further configure the appender. The following properties are available:

Property	Default Value	Description
`projectId`	If not set, default value is determined in the following order: 1. `SPRING_CLOUD_GCP_LOGGING_PROJECT_ID` Environmental Variable. 2. Value of `DefaultGcpProjectIdProvider.getProjectId()`	This is used to generate fully qualified Stackdriver Trace ID format: `projects/[PROJECT-ID]/traces/[TRACE-ID]`. This format is required to correlate trace between Stackdriver Trace and Stackdriver Logging. If `projectId` is not set and cannot be determined, then it'll log `traceId` without the fully qualified format.
`includeTraceId`	`true`	Should the `traceId` be included
`includeSpanId`	`true`	Should the `spanId` be included
`includeLevel`	`true`	Should the severity be included
`includeThreadName`	`true`	Should the thread name be included

Property	Default Value	Description
`includeMDC`	`true`	Should all MDC properties be included. The MDC properties `X-B3-TraceId`, `X-B3-SpanId` and `X-Span-Export` provided by Spring Sleuth will get excluded as they get handled separately
`includeLoggerName`	`true`	Should the name of the logger be included
`includeFormattedMessage`	`true`	Should the formatted log message be included.
`includeExceptionInMessage`	`true`	Should the stacktrace be appended to the formatted log message. This setting is only evaluated if `includeFormattedMessage` is `true`
`includeContextName`	`true`	Should the logging context be included
`includeMessage`	`false`	Should the log message with blank placeholders be included
`includeException`	`false`	Should the stacktrace be included as a own field
`serviceContext`	none	Define the Stackdriver service context data (service and version). This allows filtering of error reports for service and version in the Google Cloud Error Reporting View.
`customJson`	none	Defines custom json data. Data will be added to the json output.

This is an example of such an Logback configuration:

```
<configuration >
  <property name="projectId" value="${projectId:-${GOOGLE_CLOUD_PROJECT}}"/>

  <appender name="CONSOLE_JSON" class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
      <layout class="org.springframework.cloud.gcp.logging.StackdriverJsonLayout">
        <projectId>${projectId}</projectId>

        <!--<includeTraceId>true</includeTraceId>-->
        <!--<includeSpanId>true</includeSpanId>-->
        <!--<includeLevel>true</includeLevel>-->
        <!--<includeThreadName>true</includeThreadName>-->
        <!--<includeMDC>true</includeMDC>-->
        <!--<includeLoggerName>true</includeLoggerName>-->
        <!--<includeFormattedMessage>true</includeFormattedMessage>-->
        <!--<includeExceptionInMessage>true</includeExceptionInMessage>-->
        <!--<includeContextName>true</includeContextName>-->
        <!--<includeMessage>false</includeMessage>-->
        <!--<includeException>false</includeException>-->
        <!--<serviceContext>
              <service>service-name</service>
              <version>service-version</version>
            </serviceContext>-->
        <!--<customJson>{"custom-key": "custom-value"}</customJson>-->
      </layout>
    </encoder>
  </appender>
</configuration>
```

98.3. Sample

A Sample Spring Boot Application is provided to show how to use the Cloud logging starter.

Chapter 99. Stackdriver Monitoring

Google Cloud Platform provides a service called Stackdriver Monitoring, and Micrometer can be used with it to easily instrument Spring Boot applications for observability.

Spring Boot already provides auto-configuration for Stackdriver. This module enables to auto-detect the `project-id` and `credentials`. Also, it can be customized.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-metrics</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
metrics'
}
```

You must enable Stackdriver Monitoring API from the Google Cloud Console in order to capture metrics. Navigate to the Stackdriver Monitoring API for your project and make sure it's enabled.

This starter requires `org.springframework.boot:spring-boot-starter-actuator` dependency to activate. Make sure the dependency is being declared.

Maven coordinates:

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

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-actuator'
}
```

99.1. Spring Boot Starter for Stackdriver Monitoring

Spring Boot Starter for Stackdriver Monitoring uses Micrometer.

All configurations are optional:

Name	Description	Required	Default value
`spring.cloud.gcp.metrics.enabled`	Auto-configure Micrometer to send metrics to Stackdriver Monitoring.	No	`true`
`spring.cloud.gcp.metrics.project-id`	Overrides the project ID from the Spring Cloud GCP Module	No	
`spring.cloud.gcp.metrics.credentials.location`	Overrides the credentials location from the Spring Cloud GCP Module	No	
`spring.cloud.gcp.metrics.credentials.encoded-key`	Overrides the credentials encoded key from the Spring Cloud GCP Module	No	
`spring.cloud.gcp.metrics.credentials.scopes`	Overrides the credentials scopes from the Spring Cloud GCP Module	No	

You can use core Spring Boot Actuator properties to control reporting frequency, etc. Read Spring Boot Actuator documentation for more information on Stackdriver Actuator configurations.

Chapter 100. Spring Data Cloud Spanner

Spring Data is an abstraction for storing and retrieving POJOs in numerous storage technologies. Spring Cloud GCP adds Spring Data support for Google Cloud Spanner.

Maven coordinates for this module only, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-data-spanner</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-data-spanner'
}
```

We provide a Spring Boot Starter for Spring Data Spanner, with which you can leverage our recommended auto-configuration setup. To use the starter, see the coordinates see below.

Maven:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-data-spanner</artifactId>
</dependency>
```

Gradle:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-data-
spanner'
}
```

This setup takes care of bringing in the latest compatible version of Cloud Java Cloud Spanner libraries as well.

100.1. Configuration

To setup Spring Data Cloud Spanner, you have to configure the following:

- Setup the connection details to Google Cloud Spanner.
- Enable Spring Data Repositories (optional).

100.1.1. Cloud Spanner settings

You can the use Spring Boot Starter for Spring Data Spanner to autoconfigure Google Cloud Spanner in your Spring application. It contains all the necessary setup that makes it easy to authenticate with your Google Cloud project. The following configuration options are available:

Name	Description	Required	Default value
`spring.cloud.gcp.spanner.instance-id`	Cloud Spanner instance to use	Yes	
`spring.cloud.gcp.spanner.database`	Cloud Spanner database to use	Yes	
`spring.cloud.gcp.spanner.project-id`	GCP project ID where the Google Cloud Spanner API is hosted, if different from the one in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.spanner.credentials.location`	OAuth2 credentials for authenticating with the Google Cloud Spanner API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.spanner.credentials.encoded-key`	Base64-encoded OAuth2 credentials for authenticating with the Google Cloud Spanner API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.spanner.credentials.scopes`	OAuth2 scope for Spring Cloud GCP Cloud Spanner credentials	No	www.googleapis.com/auth/spanner.data
`spring.cloud.gcp.spanner.createInterleavedTableDdlOnDeleteCascade`	If `true`, then schema statements generated by `SpannerSchemaUtils` for tables with interleaved parent-child relationships will be "ON DELETE CASCADE". The schema for the tables will be "ON DELETE NO ACTION" if `false`.	No	`true`

`spring.cloud.gcp.spanner.numRpcChannels`	Number of gRPC channels used to connect to Cloud Spanner	No	4 - Determined by Cloud Spanner client library
`spring.cloud.gcp.spanner.prefetchChunks`	Number of chunks prefetched by Cloud Spanner for read and query	No	4 - Determined by Cloud Spanner client library
`spring.cloud.gcp.spanner.minSessions`	Minimum number of sessions maintained in the session pool	No	0 - Determined by Cloud Spanner client library
`spring.cloud.gcp.spanner.maxSessions`	Maximum number of sessions session pool can have	No	400 - Determined by Cloud Spanner client library
`spring.cloud.gcp.spanner.maxIdleSessions`	Maximum number of idle sessions session pool will maintain	No	0 - Determined by Cloud Spanner client library
`spring.cloud.gcp.spanner.writeSessionsFraction`	Fraction of sessions to be kept prepared for write transactions	No	0.2 - Determined by Cloud Spanner client library
`spring.cloud.gcp.spanner.keepAliveIntervalMinutes`	How long to keep idle sessions alive	No	30 - Determined by Cloud Spanner client library
`spring.cloud.gcp.spanner.failIfPoolExhausted`	If all sessions are in use, fail the request by throwing an exception. Otherwise, by default, block until a session becomes available.	No	`false`
`spring.cloud.gcp.spanner.emulator.enabled`	Enables the usage of an emulator. If this is set to true, then you should set the `spring.cloud.gcp.spanner.emulator-host` to the host:port of your locally running emulator instance.	No	`false`

spring.cloud.gcp.spanner.emulator-host	The host and port of the Spanner emulator; can be overridden to specify connecting to an already-running Spanner emulator instance.	No	localhost:9010

100.1.2. Repository settings

Spring Data Repositories can be configured via the `@EnableSpannerRepositories` annotation on your main `@Configuration` class. With our Spring Boot Starter for Spring Data Cloud Spanner, `@EnableSpannerRepositories` is automatically added. It is not required to add it to any other class, unless there is a need to override finer grain configuration parameters provided by `@EnableSpannerRepositories`.

100.1.3. Autoconfiguration

Our Spring Boot autoconfiguration creates the following beans available in the Spring application context:

- an instance of `SpannerTemplate`
- an instance of `SpannerDatabaseAdminTemplate` for generating table schemas from object hierarchies and creating and deleting tables and databases
- an instance of all user-defined repositories extending `SpannerRepository`, `CrudRepository`, `PagingAndSortingRepository`, when repositories are enabled
- an instance of `DatabaseClient` from the Google Cloud Java Client for Spanner, for convenience and lower level API access

100.2. Object Mapping

Spring Data Cloud Spanner allows you to map domain POJOs to Cloud Spanner tables via annotations:

```
@Table(name = "traders")
public class Trader {

    @PrimaryKey
    @Column(name = "trader_id")
    String traderId;

    String firstName;

    String lastName;

    @NotMapped
    Double temporaryNumber;
}
```

Spring Data Cloud Spanner will ignore any property annotated with `@NotMapped`. These properties will not be written to or read from Spanner.

100.2.1. Constructors

Simple constructors are supported on POJOs. The constructor arguments can be a subset of the persistent properties. Every constructor argument needs to have the same name and type as a persistent property on the entity and the constructor should set the property from the given argument. Arguments that are not directly set to properties are not supported.

```
@Table(name = "traders")
public class Trader {
    @PrimaryKey
    @Column(name = "trader_id")
    String traderId;

    String firstName;

    String lastName;

    @NotMapped
    Double temporaryNumber;

    public Trader(String traderId, String firstName) {
        this.traderId = traderId;
        this.firstName = firstName;
    }
}
```

100.2.2. Table

The `@Table` annotation can provide the name of the Cloud Spanner table that stores instances of the annotated class, one per row. This annotation is optional, and if not given, the name of the table is

inferred from the class name with the first character uncapitalized.

SpEL expressions for table names

In some cases, you might want the `@Table` table name to be determined dynamically. To do that, you can use Spring Expression Language.

For example:

```
@Table(name = "trades_#{tableNameSuffix}")
public class Trade {
    // ...
}
```

The table name will be resolved only if the `tableNameSuffix` value/bean in the Spring application context is defined. For example, if `tableNameSuffix` has the value "123", the table name will resolve to `trades_123`.

100.2.3. Primary Keys

For a simple table, you may only have a primary key consisting of a single column. Even in that case, the `@PrimaryKey` annotation is required. `@PrimaryKey` identifies the one or more ID properties corresponding to the primary key.

Spanner has first class support for composite primary keys of multiple columns. You have to annotate all of your POJO's fields that the primary key consists of with `@PrimaryKey` as below:

```
@Table(name = "trades")
public class Trade {
    @PrimaryKey(keyOrder = 2)
    @Column(name = "trade_id")
    private String tradeId;

    @PrimaryKey(keyOrder = 1)
    @Column(name = "trader_id")
    private String traderId;

    private String action;

    private Double price;

    private Double shares;

    private String symbol;
}
```

The `keyOrder` parameter of `@PrimaryKey` identifies the properties corresponding to the primary key columns in order, starting with 1 and increasing consecutively. Order is important and must reflect

the order defined in the Cloud Spanner schema. In our example the DDL to create the table and its primary key is as follows:

```
CREATE TABLE trades (
    trader_id STRING(MAX),
    trade_id STRING(MAX),
    action STRING(15),
    symbol STRING(10),
    price FLOAT64,
    shares FLOAT64
) PRIMARY KEY (trader_id, trade_id)
```

Spanner does not have automatic ID generation. For most use-cases, sequential IDs should be used with caution to avoid creating data hotspots in the system. Read Spanner Primary Keys documentation for a better understanding of primary keys and recommended practices.

100.2.4. Columns

All accessible properties on POJOs are automatically recognized as a Cloud Spanner column. Column naming is generated by the `PropertyNameFieldNamingStrategy` by default defined on the `SpannerMappingContext` bean. The `@Column` annotation optionally provides a different column name than that of the property and some other settings:

- `name` is the optional name of the column
- `spannerTypeMaxLength` specifies for `STRING` and `BYTES` columns the maximum length. This setting is only used when generating DDL schema statements based on domain types.
- `nullable` specifies if the column is created as `NOT NULL`. This setting is only used when generating DDL schema statements based on domain types.
- `spannerType` is the Cloud Spanner column type you can optionally specify. If this is not specified then a compatible column type is inferred from the Java property type.
- `spannerCommitTimestamp` is a boolean specifying if this property corresponds to an auto-populated commit timestamp column. Any value set in this property will be ignored when writing to Cloud Spanner.

100.2.5. Embedded Objects

If an object of type `B` is embedded as a property of `A`, then the columns of `B` will be saved in the same Cloud Spanner table as those of `A`.

If `B` has primary key columns, those columns will be included in the primary key of `A`. `B` can also have embedded properties. Embedding allows reuse of columns between multiple entities, and can be useful for implementing parent-child situations, because Cloud Spanner requires child tables to include the key columns of their parents.

For example:

```
class X {
  @PrimaryKey
  String grandParentId;

  long age;
}

class A {
  @PrimaryKey
  @Embedded
  X grandParent;

  @PrimaryKey(keyOrder = 2)
  String parentId;

  String value;
}

@Table(name = "items")
class B {
  @PrimaryKey
  @Embedded
  A parent;

  @PrimaryKey(keyOrder = 2)
  String id;

  @Column(name = "child_value")
  String value;
}
```

Entities of `B` can be stored in a table defined as:

```
CREATE TABLE items (
    grandParentId STRING(MAX),
    parentId STRING(MAX),
    id STRING(MAX),
    value STRING(MAX),
    child_value STRING(MAX),
    age INT64
) PRIMARY KEY (grandParentId, parentId, id)
```

Note that embedded properties' column names must all be unique.

100.2.6. Relationships

Spring Data Cloud Spanner supports parent-child relationships using the Cloud Spanner parent-child interleaved table mechanism. Cloud Spanner interleaved tables enforce the one-to-many

relationship and provide efficient queries and operations on entities of a single domain parent entity. These relationships can be up to 7 levels deep. Cloud Spanner also provides automatic cascading delete or enforces the deletion of child entities before parents.

While one-to-one and many-to-many relationships can be implemented in Cloud Spanner and Spring Data Cloud Spanner using constructs of interleaved parent-child tables, only the parent-child relationship is natively supported. Cloud Spanner does not support the foreign key constraint, though the parent-child key constraint enforces a similar requirement when used with interleaved tables.

For example, the following Java entities:

```
@Table(name = "Singers")
class Singer {
  @PrimaryKey
  long SingerId;

  String FirstName;

  String LastName;

  byte[] SingerInfo;

  @Interleaved
  List<Album> albums;
}

@Table(name = "Albums")
class Album {
  @PrimaryKey
  long SingerId;

  @PrimaryKey(keyOrder = 2)
  long AlbumId;

  String AlbumTitle;
}
```

These classes can correspond to an existing pair of interleaved tables. The `@Interleaved` annotation may be applied to `Collection` properties and the inner type is resolved as the child entity type. The schema needed to create them can also be generated using the `SpannerSchemaUtils` and run by using the `SpannerDatabaseAdminTemplate`:

```
@Autowired
SpannerSchemaUtils schemaUtils;

@Autowired
SpannerDatabaseAdminTemplate databaseAdmin;
...

// Get the create statmenets for all tables in the table structure rooted at Singer
List<String> createStrings =
this.schemaUtils.getCreateTableDdlStringsForInterleavedHierarchy(Singer.class);

// Create the tables and also create the database if necessary
this.databaseAdmin.executeDdlStrings(createStrings, true);
```

The `createStrings` list contains table schema statements using column names and types compatible with the provided Java type and any resolved child relationship types contained within based on the configured custom converters.

```
CREATE TABLE Singers (
  SingerId   INT64 NOT NULL,
  FirstName  STRING(1024),
  LastName   STRING(1024),
  SingerInfo BYTES(MAX),
) PRIMARY KEY (SingerId);

CREATE TABLE Albums (
  SingerId     INT64 NOT NULL,
  AlbumId      INT64 NOT NULL,
  AlbumTitle   STRING(MAX),
) PRIMARY KEY (SingerId, AlbumId),
  INTERLEAVE IN PARENT Singers ON DELETE CASCADE;
```

The `ON DELETE CASCADE` clause indicates that Cloud Spanner will delete all Albums of a singer if the Singer is deleted. The alternative is `ON DELETE NO ACTION`, where a Singer cannot be deleted until all of its Albums have already been deleted. When using `SpannerSchemaUtils` to generate the schema strings, the `spring.cloud.gcp.spanner.createInterleavedTableDdlOnDeleteCascade` boolean setting determines if these schema are generated as `ON DELETE CASCADE` for `true` and `ON DELETE NO ACTION` for `false`.

Cloud Spanner restricts these relationships to 7 child layers. A table may have multiple child tables.

On updating or inserting an object to Cloud Spanner, all of its referenced children objects are also updated or inserted in the same request, respectively. On read, all of the interleaved child rows are also all read.

Lazy Fetch

`@Interleaved` properties are retrieved eagerly by default, but can be fetched lazily for performance

in both read and write:

```
@Interleaved(lazy = true)
List<Album> albums;
```

Lazily-fetched interleaved properties are retrieved upon the first interaction with the property. If a property marked for lazy fetching is never retrieved, then it is also skipped when saving the parent entity.

If used inside a transaction, subsequent operations on lazily-fetched properties use the same transaction context as that of the original parent entity.

Declarative Filtering with `@Where`

The `@Where` annotation could be applied to an entity class or to an interleaved property. This annotation provides an SQL where clause that will be applied at the fetching of interleaved collections or the entity itself.

Let’s say we have an `Agreement` with a list of `Participants` which could be assigned to it. We would like to fetch a list of currently active participants. For security reasons, all records should remain in the database forever, even if participants become inactive. That can be easily achieved with the `@Where` annotation, which is demonstrated by this example:

```
@Table(name = "participants")
public class Participant {
  //...
  boolean active;
  //...
}

@Table(name = "agreements")
public class Agreement {
  //...
  @Interleaved
  @Where("active = true")
  List<Participant> participants;
  Person person;
  //...
}
```

100.2.7. Supported Types

Spring Data Cloud Spanner natively supports the following types for regular fields but also utilizes custom converters (detailed in following sections) and dozens of pre-defined Spring Data custom converters to handle other common Java types.

Natively supported types:

- `com.google.cloud.ByteArray`
- `com.google.cloud.Date`
- `com.google.cloud.Timestamp`
- `java.lang.Boolean`, `boolean`
- `java.lang.Double`, `double`
- `java.lang.Long`, `long`
- `java.lang.Integer`, `int`
- `java.lang.String`
- `double[]`
- `long[]`
- `boolean[]`
- `java.util.Date`
- `java.time.Instant`
- `java.sql.Date`
- `java.time.LocalDate`
- `java.time.LocalDateTime`

100.2.8. Lists

Spanner supports `ARRAY` types for columns. `ARRAY` columns are mapped to `List` fields in POJOS.

Example:

```
List<Double> curve;
```

The types inside the lists can be any singular property type.

100.2.9. Lists of Structs

Cloud Spanner queries can construct STRUCT values that appear as columns in the result. Cloud Spanner requires STRUCT values appear in ARRAYs at the root level: `SELECT ARRAY(SELECT STRUCT(1 as val1, 2 as val2)) as pair FROM Users`.

Spring Data Cloud Spanner will attempt to read the column STRUCT values into a property that is an `Iterable` of an entity type compatible with the schema of the column STRUCT value.

For the previous array-select example, the following property can be mapped with the constructed `ARRAY<STRUCT>` column: `List<TwoInts> pair;` where the `TwoInts` type is defined:

```
class TwoInts {

  int val1;

  int val2;
}
```

100.2.10. Custom types

Custom converters can be used to extend the type support for user defined types.

1. Converters need to implement the `org.springframework.core.convert.converter.Converter` interface in both directions.
2. The user defined type needs to be mapped to one of the basic types supported by Spanner:
    - `com.google.cloud.ByteArray`
    - `com.google.cloud.Date`
    - `com.google.cloud.Timestamp`
    - `java.lang.Boolean`, `boolean`
    - `java.lang.Double`, `double`
    - `java.lang.Long`, `long`
    - `java.lang.String`
    - `double[]`
    - `long[]`
    - `boolean[]`
    - `enum` types
3. An instance of both Converters needs to be passed to a `ConverterAwareMappingSpannerEntityProcessor`, which then has to be made available as a `@Bean` for `SpannerEntityProcessor`.

For example:

We would like to have a field of type `Person` on our `Trade` POJO:

```
@Table(name = "trades")
public class Trade {
  //...
  Person person;
  //...
}
```

Where Person is a simple class:

```
public class Person {

  public String firstName;
  public String lastName;

}
```

We have to define the two converters:

```
  public class PersonWriteConverter implements Converter<Person, String> {

    @Override
    public String convert(Person person) {
      return person.firstName + " " + person.lastName;
    }
  }

  public class PersonReadConverter implements Converter<String, Person> {

    @Override
    public Person convert(String s) {
      Person person = new Person();
      person.firstName = s.split(" ")[0];
      person.lastName = s.split(" ")[1];
      return person;
    }
  }
```

That will be configured in our `@Configuration` file:

```
@Configuration
public class ConverterConfiguration {

    @Bean
    public SpannerEntityProcessor spannerEntityProcessor(SpannerMappingContext
spannerMappingContext) {
        return new ConverterAwareMappingSpannerEntityProcessor(spannerMappingContext,
                Arrays.asList(new PersonWriteConverter()),
                Arrays.asList(new PersonReadConverter()));
    }
}
```

100.2.11. Custom Converter for Struct Array Columns

If a `Converter<Struct, A>` is provided, then properties of type `List<A>` can be used in your entity types.

100.3. Spanner Operations & Template

`SpannerOperations` and its implementation, `SpannerTemplate`, provides the Template pattern familiar to Spring developers. It provides:

- Resource management
- One-stop-shop to Spanner operations with the Spring Data POJO mapping and conversion features
- Exception conversion

Using the `autoconfigure` provided by our Spring Boot Starter for Spanner, your Spring application context will contain a fully configured `SpannerTemplate` object that you can easily autowire in your application:

```
@SpringBootApplication
public class SpannerTemplateExample {

    @Autowired
    SpannerTemplate spannerTemplate;

    public void doSomething() {
        this.spannerTemplate.delete(Trade.class, KeySet.all());
        //...
        Trade t = new Trade();
        //...
        this.spannerTemplate.insert(t);
        //...
        List<Trade> tradesByAction = spannerTemplate.findAll(Trade.class);
        //...
    }
}
```

The Template API provides convenience methods for:

- Reads, and by providing SpannerReadOptions and SpannerQueryOptions
  - Stale read
  - Read with secondary indices
  - Read with limits and offsets
  - Read with sorting
- Queries
- DML operations (delete, insert, update, upsert)
- Partial reads
  - You can define a set of columns to be read into your entity
- Partial writes

    - Persist only a few properties from your entity
- Read-only transactions
- Locking read-write transactions

100.3.1. SQL Query

Cloud Spanner has SQL support for running read-only queries. All the query related methods start with `query` on `SpannerTemplate`. By using `SpannerTemplate`, you can run SQL queries that map to POJOs:

```
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT *
FROM trades"));
```

100.3.2. Read

Spanner exposes a Read API for reading single row or multiple rows in a table or in a secondary index.

Using `SpannerTemplate` you can run reads, as the following example shows:

```
List<Trade> trades = this.spannerTemplate.readAll(Trade.class);
```

Main benefit of reads over queries is reading multiple rows of a certain pattern of keys is much easier using the features of the `KeySet` class.

100.3.3. Advanced reads

Stale read

All reads and queries are **strong reads** by default. A **strong read** is a read at a current time and is guaranteed to see all data that has been committed up until the start of this read. An **exact staleness read** is read at a timestamp in the past. Cloud Spanner allows you to determine how current the data should be when you read data. With `SpannerTemplate` you can specify the `Timestamp` by setting it on `SpannerQueryOptions` or `SpannerReadOptions` to the appropriate read or query methods:

Reads:

```
// a read with options:
SpannerReadOptions spannerReadOptions = new
SpannerReadOptions().setTimestamp(myTimestamp);
List<Trade> trades = this.spannerTemplate.readAll(Trade.class, spannerReadOptions);
```

Queries:

```
// a query with options:
SpannerQueryOptions spannerQueryOptions = new
SpannerQueryOptions().setTimestamp(myTimestamp);
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT *
FROM trades"), spannerQueryOptions);
```

You can also read with **bounded staleness** by setting `.setTimestampBound(TimestampBound.ofMinReadTimestamp(myTimestamp))` on the query and read options objects. Bounded staleness lets Cloud Spanner choose any point in time later than or equal to the given timestampBound, but it cannot be used inside transactions.

Read from a secondary index

Using a secondary index is available for Reads via the Template API and it is also implicitly available via SQL for Queries.

The following shows how to read rows from a table using a secondary index simply by setting `index` on `SpannerReadOptions`:

```
SpannerReadOptions spannerReadOptions = new
SpannerReadOptions().setIndex("TradesByTrader");
List<Trade> trades = this.spannerTemplate.readAll(Trade.class, spannerReadOptions);
```

Read with offsets and limits

Limits and offsets are only supported by Queries. The following will get only the first two rows of the query:

```
SpannerQueryOptions spannerQueryOptions = new
SpannerQueryOptions().setLimit(2).setOffset(3);
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT *
FROM trades"), spannerQueryOptions);
```

Note that the above is equivalent of running `SELECT * FROM trades LIMIT 2 OFFSET 3`.

Sorting

Reads by keys do not support sorting. However, queries on the Template API support sorting through standard SQL and also via Spring Data Sort API:

```
List<Trade> trades = this.spannerTemplate.queryAll(Trade.class, Sort.by("action"));
```

If the provided sorted field name is that of a property of the domain type, then the column name corresponding to that property will be used in the query. Otherwise, the given field name is assumed to be the name of the column in the Cloud Spanner table. Sorting on columns of Cloud Spanner types STRING and BYTES can be done while ignoring case:

```
Sort.by(Order.desc("action").ignoreCase())
```

Partial read

Partial read is only possible when using Queries. In case the rows returned by the query have fewer columns than the entity that it will be mapped to, Spring Data will map the returned columns only. This setting also applies to nested structs and their corresponding nested POJO properties.

```
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT
action, symbol FROM trades"),
    new SpannerQueryOptions().setAllowMissingResultSetColumns(true));
```

If the setting is set to `false`, then an exception will be thrown if there are missing columns in the query result.

Summary of options for Query vs Read

Feature	Query supports it	Read supports it
SQL	yes	no
Partial read	yes	no
Limits	yes	no
Offsets	yes	no
Secondary index	yes	yes
Read using index range	no	yes
Sorting	yes	no

100.3.4. Write / Update

The write methods of `SpannerOperations` accept a POJO and writes all of its properties to Spanner. The corresponding Spanner table and entity metadata is obtained from the given object's actual type.

If a POJO was retrieved from Spanner and its primary key properties values were changed and then written or updated, the operation will occur as if against a row with the new primary key values. The row with the original primary key values will not be affected.

Insert

The `insert` method of `SpannerOperations` accepts a POJO and writes all of its properties to Spanner, which means the operation will fail if a row with the POJO's primary key already exists in the table.

```
Trade t = new Trade();
this.spannerTemplate.insert(t);
```

Update

The `update` method of `SpannerOperations` accepts a POJO and writes all of its properties to Spanner, which means the operation will fail if the POJO's primary key does not already exist in the table.

```
// t was retrieved from a previous operation
this.spannerTemplate.update(t);
```

Upsert

The `upsert` method of `SpannerOperations` accepts a POJO and writes all of its properties to Spanner using update-or-insert.

```
// t was retrieved from a previous operation or it's new
this.spannerTemplate.upsert(t);
```

Partial Update

The update methods of `SpannerOperations` operate by default on all properties within the given object, but also accept `String[]` and `Optional<Set<String>>` of column names. If the `Optional` of set of column names is empty, then all columns are written to Spanner. However, if the Optional is occupied by an empty set, then no columns will be written.

```
// t was retrieved from a previous operation or it's new
this.spannerTemplate.update(t, "symbol", "action");
```

100.3.5. DML

DML statements can be run by using `SpannerOperations.executeDmlStatement`. Inserts, updates, and deletions can affect any number of rows and entities.

You can run partitioned DML updates by using the `executePartitionedDmlStatement` method. Partitioned DML queries have performance benefits but also have restrictions and cannot be used inside transactions.

100.3.6. Transactions

`SpannerOperations` provides methods to run `java.util.Function` objects within a single transaction while making available the read and write methods from `SpannerOperations`.

Read/Write Transaction

Read and write transactions are provided by `SpannerOperations` via the `performReadWriteTransaction` method:

```
@Autowired
SpannerOperations mySpannerOperations;

public String doWorkInsideTransaction() {
  return mySpannerOperations.performReadWriteTransaction(
    transActionSpannerOperations -> {
      // Work with transActionSpannerOperations here.
      // It is also a SpannerOperations object.

      return "transaction completed";
    }
  );
}
```

The `performReadWriteTransaction` method accepts a `Function` that is provided an instance of a `SpannerOperations` object. The final returned value and type of the function is determined by the user. You can use this object just as you would a regular `SpannerOperations` with a few exceptions:

- Its read functionality cannot perform stale reads, because all reads and writes happen at the single point in time of the transaction.
- It cannot perform sub-transactions via `performReadWriteTransaction` or `performReadOnlyTransaction`.

As these read-write transactions are locking, it is recommended that you use the `performReadOnlyTransaction` if your function does not perform any writes.

Read-only Transaction

The `performReadOnlyTransaction` method is used to perform read-only transactions using a `SpannerOperations`:

```
@Autowired
SpannerOperations mySpannerOperations;

public String doWorkInsideTransaction() {
  return mySpannerOperations.performReadOnlyTransaction(
    transActionSpannerOperations -> {
      // Work with transActionSpannerOperations here.
      // It is also a SpannerOperations object.

      return "transaction completed";
    }
  );
}
```

The `performReadOnlyTransaction` method accepts a `Function` that is provided an instance of a `SpannerOperations` object. This method also accepts a `ReadOptions` object, but the only attribute used is the timestamp used to determine the snapshot in time to perform the reads in the transaction. If

the timestamp is not set in the read options the transaction is run against the current state of the database. The final returned value and type of the function is determined by the user. You can use this object just as you would a regular `SpannerOperations` with a few exceptions:

- Its read functionality cannot perform stale reads (other than the staleness set on the entire transaction), because all reads happen at the single point in time of the transaction.
- It cannot perform sub-transactions via `performReadWriteTransaction` or `performReadOnlyTransaction`
- It cannot perform any write operations.

Because read-only transactions are non-locking and can be performed on points in time in the past, these are recommended for functions that do not perform write operations.

Declarative Transactions with @Transactional Annotation

This feature requires a bean of `SpannerTransactionManager`, which is provided when using `spring-cloud-gcp-starter-data-spanner`.

`SpannerTemplate` and `SpannerRepository` support running methods with the `@Transactional` annotation as transactions. If a method annotated with `@Transactional` calls another method also annotated, then both methods will work within the same transaction. `performReadOnlyTransaction` and `performReadWriteTransaction` cannot be used in `@Transactional` annotated methods because Cloud Spanner does not support transactions within transactions.

100.3.7. DML Statements

`SpannerTemplate` supports DML `Statements`. DML statements can also be run in transactions by using `performReadWriteTransaction` or by using the `@Transactional` annotation.

100.4. Repositories

Spring Data Repositories are a powerful abstraction that can save you a lot of boilerplate code.

For example:

```
public interface TraderRepository extends SpannerRepository<Trader, String> {
}
```

Spring Data generates a working implementation of the specified interface, which can be conveniently autowired into an application.

The `Trader` type parameter to `SpannerRepository` refers to the underlying domain type. The second type parameter, `String` in this case, refers to the type of the key of the domain type.

For POJOs with a composite primary key, this ID type parameter can be any descendant of `Object[]` compatible with all primary key properties, any descendant of `Iterable`, or `com.google.cloud.spanner.Key`. If the domain POJO type only has a single primary key column, then the primary key property type can be used or the `Key` type.

For example in case of Trades, that belong to a Trader, `TradeRepository` would look like this:

```
public interface TradeRepository extends SpannerRepository<Trade, String[]> {

}
```

```
public class MyApplication {

    @Autowired
    SpannerTemplate spannerTemplate;

    @Autowired
    StudentRepository studentRepository;

    public void demo() {

        this.tradeRepository.deleteAll();
        String traderId = "demo_trader";
        Trade t = new Trade();
        t.symbol = stock;
        t.action = action;
        t.traderId = traderId;
        t.price = 100.0;
        t.shares = 12345.6;
        this.spannerTemplate.insert(t);

        Iterable<Trade> allTrades = this.tradeRepository.findAll();

        int count = this.tradeRepository.countByAction("BUY");

    }
}
```

100.4.1. CRUD Repository

`CrudRepository` methods work as expected, with one thing Spanner specific: the `save` and `saveAll` methods work as update-or-insert.

100.4.2. Paging and Sorting Repository

You can also use `PagingAndSortingRepository` with Spanner Spring Data. The sorting and pageable `findAll` methods available from this interface operate on the current state of the Spanner database. As a result, beware that the state of the database (and the results) might change when moving page to page.

100.4.3. Spanner Repository

The `SpannerRepository` extends the `PagingAndSortingRepository`, but adds the read-only and the read-write transaction functionality provided by Spanner. These transactions work very similarly to those of `SpannerOperations`, but is specific to the repository's domain type and provides repository functions instead of template functions.

For example, this is a read-only transaction:

```
@Autowired
SpannerRepository myRepo;

public String doWorkInsideTransaction() {
  return myRepo.performReadOnlyTransaction(
    transactionSpannerRepo -> {
      // Work with the single-transaction transactionSpannerRepo here.
      // This is a SpannerRepository object.

      return "transaction completed";
    }
  );
}
```

When creating custom repositories for your own domain types and query methods, you can extend `SpannerRepository` to access Cloud Spanner-specific features as well as all features from `PagingAndSortingRepository` and `CrudRepository`.

100.5. Query Methods

`SpannerRepository` supports Query Methods. Described in the following sections, these are methods residing in your custom repository interfaces of which implementations are generated based on their names and annotations. Query Methods can read, write, and delete entities in Cloud Spanner. Parameters to these methods can be any Cloud Spanner data type supported directly or via custom configured converters. Parameters can also be of type `Struct` or POJOs. If a POJO is given as a parameter, it will be converted to a `Struct` with the same type-conversion logic as used to create write mutations. Comparisons using Struct parameters are limited to what is available with Cloud Spanner.

100.5.1. Query methods by convention

```
public interface TradeRepository extends SpannerRepository<Trade, String[]> {
    List<Trade> findByAction(String action);

    int countByAction(String action);

    // Named methods are powerful, but can get unwieldy
    List<Trade>
findTop3DistinctByActionAndSymbolIgnoreCaseOrTraderIdOrderBySymbolDesc(
            String action, String symbol, String traderId);
}
```

In the example above, the query methods in `TradeRepository` are generated based on the name of the methods, using the Spring Data Query creation naming convention.

`List<Trade> findByAction(String action)` would translate to a `SELECT * FROM trades WHERE action = ?`.

The function `List<Trade> findTop3DistinctByActionAndSymbolIgnoreCaseOrTraderIdOrderBySymbolDesc(String action, String symbol, String traderId);` will be translated as the equivalent of this SQL query:

```
SELECT DISTINCT * FROM trades
WHERE ACTION = ? AND LOWER(SYMBOL) = LOWER(?) AND TRADER_ID = ?
ORDER BY SYMBOL DESC
LIMIT 3
```

The following filter options are supported:

- Equality
- Greater than or equals
- Greater than
- Less than or equals
- Less than
- Is null
- Is not null
- Is true
- Is false
- Like a string
- Not like a string
- Contains a string
- Not contains a string
- In

- Not in

Note that the phrase `SymbolIgnoreCase` is translated to `LOWER(SYMBOL) = LOWER(?)` indicating a non-case-sensitive matching. The `IgnoreCase` phrase may only be appended to fields that correspond to columns of type STRING or BYTES. The Spring Data "AllIgnoreCase" phrase appended at the end of the method name is not supported.

The `Like` or `NotLike` naming conventions:

```
List<Trade> findBySymbolLike(String symbolFragment);
```

The param `symbolFragment` can contain wildcard characters for string matching such as `_` and `%`.

The `Contains` and `NotContains` naming conventions:

```
List<Trade> findBySymbolContains(String symbolFragment);
```

The param `symbolFragment` is a regular expression that is checked for occurrences.

The `In` and `NotIn` keywords must be used with `Iterable` corresponding parameters.

Delete queries are also supported. For example, query methods such as `deleteByAction` or `removeByAction` delete entities found by `findByAction`. The delete operation happens in a single transaction.

Delete queries can have the following return types: * An integer type that is the number of entities deleted * A collection of entities that were deleted * `void`

100.5.2. Custom SQL/DML query methods

The example above for `List<Trade> fetchByActionNamedQuery(String action)` does not match the Spring Data Query creation naming convention, so we have to map a parametrized Spanner SQL query to it.

The SQL query for the method can be mapped to repository methods in one of two ways:

- `namedQueries` properties file
- using the `@Query` annotation

The names of the tags of the SQL correspond to the `@Param` annotated names of the method parameters.

Interleaved properties are loaded eagerly, unless they are annotated with `@Interleaved(lazy = true)`.

Custom SQL query methods can accept a single `Sort` or `Pageable` parameter that is applied on top of the specified custom query. It is the recommended way to control the sort order of the results, which is not guaranteed by the `ORDER BY` clause in the SQL query. This is due to the fact that the

user-provided query is used as a sub-query, and Cloud Spanner doesn't preserve order in subquery results.

You might want to use `ORDER BY` with `LIMIT` to obtain the top records, according to a specified order. However, to ensure the correct sort order of the final result set, sort options have to be passed in with a `Pageable`.

```
@Query("SELECT * FROM trades")
List<Trade> fetchTrades(Pageable pageable);

@Query("SELECT * FROM trades ORDER BY price DESC LIMIT 1")
Trade topTrade(Pageable pageable);
```

This can be used:

```
List<Trade> customSortedTrades = tradeRepository.fetchTrades(PageRequest
            .of(2, 2, org.springframework.data.domain.Sort.by(Order.asc("id"))));
```

The results would be sorted by "id" in ascending order.

Your query method can also return non-entity types:

```
@Query("SELECT COUNT(1) FROM trades WHERE action = @action")
int countByActionQuery(String action);

@Query("SELECT EXISTS(SELECT COUNT(1) FROM trades WHERE action = @action)")
boolean existsByActionQuery(String action);

@Query("SELECT action FROM trades WHERE action = @action LIMIT 1")
String getFirstString(@Param("action") String action);

@Query("SELECT action FROM trades WHERE action = @action")
List<String> getFirstStringList(@Param("action") String action);
```

DML statements can also be run by query methods, but the only possible return value is a `long` representing the number of affected rows. The `dmlStatement` boolean setting must be set on `@Query` to indicate that the query method is run as a DML statement.

```
@Query(value = "DELETE FROM trades WHERE action = @action", dmlStatement = true)
long deleteByActionQuery(String action);
```

Query methods with named queries properties

By default, the `namedQueriesLocation` attribute on `@EnableSpannerRepositories` points to the `META-INF/spanner-named-queries.properties` file. You can specify the query for a method in the properties file by providing the SQL as the value for the "interface.method" property:

```
Trade.fetchByActionNamedQuery=SELECT * FROM trades WHERE trades.action = @tag0
```

```
public interface TradeRepository extends SpannerRepository<Trade, String[]> {
    // This method uses the query from the properties file instead of one generated
based on name.
    List<Trade> fetchByActionNamedQuery(@Param("tag0") String action);
}
```

Query methods with annotation

Using the `@Query` annotation:

```
public interface TradeRepository extends SpannerRepository<Trade, String[]> {
    @Query("SELECT * FROM trades WHERE trades.action = @tag0")
    List<Trade> fetchByActionNamedQuery(@Param("tag0") String action);
}
```

Table names can be used directly. For example, "trades" in the above example. Alternatively, table names can be resolved from the `@Table` annotation on domain classes as well. In this case, the query should refer to table names with fully qualified class names between `:` characters: `:fully.qualified.ClassName:`. A full example would look like:

```
@Query("SELECT * FROM :com.example.Trade: WHERE trades.action = @tag0")
List<Trade> fetchByActionNamedQuery(String action);
```

This allows table names evaluated with SpEL to be used in custom queries.

SpEL can also be used to provide SQL parameters:

```
@Query("SELECT * FROM :com.example.Trade: WHERE trades.action = @tag0
  AND price > #{#priceRadius * -1} AND price < #{#priceRadius * 2}")
List<Trade> fetchByActionNamedQuery(String action, Double priceRadius);
```

When using the `IN` SQL clause, remember to use `IN UNNEST(@iterableParam)` to specify a single `Iterable` parameter. You can also use a fixed number of singular parameters such as `IN (@stringParam1, @stringParam2)`.

100.5.3. Projections

Spring Data Spanner supports projections. You can define projection interfaces based on domain types and add query methods that return them in your repository:

```
public interface TradeProjection {

    String getAction();

    @Value("#{target.symbol + ' ' + target.action}")
    String getSymbolAndAction();
}

public interface TradeRepository extends SpannerRepository<Trade, Key> {

    List<Trade> findByTraderId(String traderId);

    List<TradeProjection> findByAction(String action);

    @Query("SELECT action, symbol FROM trades WHERE action = @action")
    List<TradeProjection> findByQuery(String action);
}
```

Projections can be provided by name-convention-based query methods as well as by custom SQL queries. If using custom SQL queries, you can further restrict the columns retrieved from Spanner to just those required by the projection to improve performance.

Properties of projection types defined using SpEL use the fixed name `target` for the underlying domain object. As a result accessing underlying properties take the form `target.<property-name>`.

100.5.4. Empty result handling in repository methods

Java `java.util.Optional` can be used to indicate the potential absence of a return value.

Alternatively, query methods can return the result without a wrapper. In that case the absence of a query result is indicated by returning `null`. Repository methods returning collections are guaranteed never to return `null` but rather the corresponding empty collection.

You can enable nullability checks. For more details please see Spring Framework's nullability docs.

100.5.5. REST Repositories

When running with Spring Boot, repositories can be exposed as REST services by simply adding this dependency to your pom file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

If you prefer to configure parameters (such as path), you can use `@RepositoryRestResource`

annotation:

```
@RepositoryRestResource(collectionResourceRel = "trades", path = "trades")
public interface TradeRepository extends SpannerRepository<Trade, Key> {
}
```

For classes that have composite keys (multiple `@PrimaryKey` fields), only the `Key` type is supported for the repository ID type.

For example, you can retrieve all `Trade` objects in the repository by using `curl http://<server>:<port>/trades`, or any specific trade via `curl http://<server>:<port>/trades/<trader_id>,<trade_id>`.

The separator between your primary key components, `id` and `trader_id` in this case, is a comma by default, but can be configured to any string not found in your key values by extending the `SpannerKeyIdConverter` class:

```
@Component
class MySpecialIdConverter extends SpannerKeyIdConverter {

    @Override
    protected String getUrlIdSeparator() {
        return ":";
    }
}
```

You can also write trades using `curl -XPOST -H"Content-Type: application/json" -d@test.json http://<server>:<port>/trades/` where the file `test.json` holds the JSON representation of a `Trade` object.

100.6. Database and Schema Admin

Databases and tables inside Spanner instances can be created automatically from `SpannerPersistentEntity` objects:

```
@Autowired
private SpannerSchemaUtils spannerSchemaUtils;

@Autowired
private SpannerDatabaseAdminTemplate spannerDatabaseAdminTemplate;

public void createTable(SpannerPersistentEntity entity) {
    if(!spannerDatabaseAdminTemplate.tableExists(entity.tableName()){

      // The boolean parameter indicates that the database will be created if it does
not exist.
      spannerDatabaseAdminTemplate.executeDdlStrings(Arrays.asList(
            spannerSchemaUtils.getCreateTableDDLString(entity.getType())), true);
    }
}
```

Schemas can be generated for entire object hierarchies with interleaved relationships and composite keys.

100.7. Events

Spring Data Cloud Spanner publishes events extending the Spring Framework's `ApplicationEvent` to the context that can be received by `ApplicationListener` beans you register.

Type	Description	Contents
`AfterReadEvent`	Published immediately after entities are read by key from Cloud Spanner by `SpannerTemplate`	The entities loaded. The read options and key-set originally specified for the load operation.
`AfterQueryEvent`	Published immediately after entities are read by query from Cloud Spanner by `SpannerTemplate`	The entities loaded. The query options and query statement originally specified for the load operation.
`BeforeExecuteDmlEvent`	Published immediately before DML statements are executed by `SpannerTemplate`	The DML statement to execute.
`AfterExecuteDmlEvent`	Published immediately after DML statements are executed by `SpannerTemplate`	The DML statement to execute and the number of rows affected by the operation as reported by Cloud Spanner.
`BeforeSaveEvent`	Published immediately before upsert/update/insert operations are executed by `SpannerTemplate`	The mutations to be sent to Cloud Spanner, the entities to be saved, and optionally the properties in those entities to save.

Type	Description	Contents
`AfterSaveEvent`	Published immediately after upsert/update/insert operations are executed by `SpannerTemplate`	The mutations sent to Cloud Spanner, the entities to be saved, and optionally the properties in those entities to save.
`BeforeDeleteEvent`	Published immediately before delete operations are executed by `SpannerTemplate`	The mutations to be sent to Cloud Spanner. The target entities, keys, or entity type originally specified for the delete operation.
`AfterDeleteEvent`	Published immediately after delete operations are executed by `SpannerTemplate`	The mutations sent to Cloud Spanner. The target entities, keys, or entity type originally specified for the delete operation.

100.8. Auditing

Spring Data Cloud Spanner supports the `@LastModifiedDate` and `@LastModifiedBy` auditing annotations for properties:

```
@Table
public class SimpleEntity {
    @PrimaryKey
    String id;

    @LastModifiedBy
    String lastUser;

    @LastModifiedDate
    DateTime lastTouched;
}
```

Upon insert, update, or save, these properties will be set automatically by the framework before mutations are generated and saved to Cloud Spanner.

To take advantage of these features, add the `@EnableSpannerAuditing` annotation to your configuration class and provide a bean for an `AuditorAware<A>` implementation where the type `A` is the desired property type annotated by `@LastModifiedBy`:

```
@Configuration
@EnableSpannerAuditing
public class Config {

    @Bean
    public AuditorAware<String> auditorProvider() {
        return () -> Optional.of("YOUR_USERNAME_HERE");
    }
}
```

The `AuditorAware` interface contains a single method that supplies the value for fields annotated by `@LastModifiedBy` and can be of any type. One alternative is to use Spring Security's `User` type:

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  public Optional<User> getCurrentAuditor() {

    return Optional.ofNullable(SecurityContextHolder.getContext())
              .map(SecurityContext::getAuthentication)
              .filter(Authentication::isAuthenticated)
              .map(Authentication::getPrincipal)
              .map(User.class::cast);
  }
}
```

You can also set a custom provider for properties annotated `@LastModifiedDate` by providing a bean for `DateTimeProvider` and providing the bean name to `@EnableSpannerAuditing(dateTimeProviderRef = "customDateTimeProviderBean")`.

100.9. Multi-Instance Usage

Your application can be configured to use multiple Cloud Spanner instances or databases by providing a custom bean for `DatabaseIdProvider`. The default bean uses the instance ID, database name, and project ID options you configured in `application.properties`.

```
    @Bean
    public DatabaseIdProvider databaseIdProvider() {
        // return custom connection options provider
    }
```

The `DatabaseId` given by this provider is used as the target database name and instance of each operation Spring Data Cloud Spanner executes. By providing a custom implementation of this bean (for example, supplying a thread-local `DatabaseId`), you can direct your application to use multiple instances or databases.

Database administrative operations, such as creating tables using `SpannerDatabaseAdminTemplate`,

will also utilize the provided `DatabaseId`.

If you would like to configure every aspect of each connection (such as pool size and retry settings), you can supply a bean for `Supplier<DatabaseClient>`.

100.10. Cloud Spanner Emulator

The Cloud SDK provides a local, in-memory emulator for Cloud Spanner, which you can use to develop and test your application. As the emulator stores data only in memory, it will not persist data across runs. It is intended to help you use Cloud Spanner for local development and testing, not for production deployments.

In order to set up and start the emulator, you can follow these steps.

This command can be used to create Cloud Spanner instances:

```
$ gcloud spanner instances create <instance-name> --config=emulator-config
--description="<description>" --nodes=1
```

Once the Spanner emulator is running, ensure that the following properties are set in your `application.properties` of your Spring application:

```
spring.cloud.gcp.spanner.emulator.enabled=true
spring.cloud.gcp.spanner.emulator-host=${EMULATOR_HOSTPORT}
```

100.11. Sample

A sample application is available.

Chapter 101. Spring Data Cloud Datastore

This integration is fully compatible with Firestore in Datastore Mode, but not with Firestore in Native Mode.

Spring Data is an abstraction for storing and retrieving POJOs in numerous storage technologies. Spring Cloud GCP adds Spring Data support for Google Cloud Firestore in Datastore mode.

Maven coordinates for this module only, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-data-datastore</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-data-
datastore'
}
```

We provide a Spring Boot Starter for Spring Data Datastore, with which you can use our recommended auto-configuration setup. To use the starter, see the coordinates below.

Maven:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-data-datastore</artifactId>
</dependency>
```

Gradle:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-data-
datastore'
}
```

This setup takes care of bringing in the latest compatible version of Cloud Java Cloud Datastore libraries as well.

101.1. Configuration

To setup Spring Data Cloud Datastore, you have to configure the following:

- Setup the connection details to Google Cloud Datastore.

101.1.1. Cloud Datastore settings

You can use the Spring Boot Starter for Spring Data Datastore to autoconfigure Google Cloud Datastore in your Spring application. It contains all the necessary setup that makes it easy to authenticate with your Google Cloud project. The following configuration options are available:

Name	Description	Required	Default value
`spring.cloud.gcp.datastore.enabled`	Enables the Cloud Datastore client	No	`true`
`spring.cloud.gcp.datastore.project-id`	GCP project ID where the Google Cloud Datastore API is hosted, if different from the one in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.datastore.credentials.location`	OAuth2 credentials for authenticating with the Google Cloud Datastore API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.datastore.credentials.encoded-key`	Base64-encoded OAuth2 credentials for authenticating with the Google Cloud Datastore API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.datastore.credentials.scopes`	OAuth2 scope for Spring Cloud GCP Cloud Datastore credentials	No	www.googleapis.com/auth/datastore
`spring.cloud.gcp.datastore.namespace`	The Cloud Datastore namespace to use	No	the Default namespace of Cloud Datastore in your GCP project

`spring.cloud.gcp.datastore.host`	The `hostname:port` of the datastore service or emulator to connect to. Can be used to connect to a manually started Datastore Emulator. If the autoconfigured emulator is enabled, this property will be ignored and `localhost:<emulator_port>` will be used.	No	
`spring.cloud.gcp.datastore.emulator.enabled`	To enable the auto configuration to start a local instance of the Datastore Emulator.	No	`false`
`spring.cloud.gcp.datastore.emulator.port`	The local port to use for the Datastore Emulator	No	`8081`
`spring.cloud.gcp.datastore.emulator.consistency`	The consistency to use for the Datastore Emulator instance	No	`0.9`

101.1.2. Repository settings

Spring Data Repositories can be configured via the `@EnableDatastoreRepositories` annotation on your main `@Configuration` class. With our Spring Boot Starter for Spring Data Cloud Datastore, `@EnableDatastoreRepositories` is automatically added. It is not required to add it to any other class, unless there is a need to override finer grain configuration parameters provided by `@EnableDatastoreRepositories`.

101.1.3. Autoconfiguration

Our Spring Boot autoconfiguration creates the following beans available in the Spring application context:

- an instance of `DatastoreTemplate`
- an instance of all user defined repositories extending `CrudRepository`, `PagingAndSortingRepository`, and `DatastoreRepository` (an extension of `PagingAndSortingRepository` with additional Cloud Datastore features) when repositories are enabled
- an instance of `Datastore` from the Google Cloud Java Client for Datastore, for convenience and lower level API access

101.1.4. Datastore Emulator Autoconfiguration

This Spring Boot autoconfiguration can also configure and start a local Datastore Emulator server if

enabled by property.

It is useful for integration testing, but not for production.

When enabled, the `spring.cloud.gcp.datastore.host` property will be ignored and the Datastore autoconfiguration itself will be forced to connect to the autoconfigured local emulator instance.

It will create an instance of `LocalDatastoreHelper` as a bean that stores the `DatastoreOptions` to get the `Datastore` client connection to the emulator for convenience and lower level API for local access. The emulator will be properly stopped after the Spring application context shutdown.

101.2. Object Mapping

Spring Data Cloud Datastore allows you to map domain POJOs to Cloud Datastore kinds and entities via annotations:

```
@Entity(name = "traders")
public class Trader {

    @Id
    @Field(name = "trader_id")
    String traderId;

    String firstName;

    String lastName;

    @Transient
    Double temporaryNumber;
}
```

Spring Data Cloud Datastore will ignore any property annotated with `@Transient`. These properties will not be written to or read from Cloud Datastore.

101.2.1. Constructors

Simple constructors are supported on POJOs. The constructor arguments can be a subset of the persistent properties. Every constructor argument needs to have the same name and type as a persistent property on the entity and the constructor should set the property from the given argument. Arguments that are not directly set to properties are not supported.

```java
@Entity(name = "traders")
public class Trader {

    @Id
    @Field(name = "trader_id")
    String traderId;

    String firstName;

    String lastName;

    @Transient
    Double temporaryNumber;

    public Trader(String traderId, String firstName) {
        this.traderId = traderId;
        this.firstName = firstName;
    }
}
```

101.2.2. Kind

The `@Entity` annotation can provide the name of the Cloud Datastore kind that stores instances of the annotated class, one per row.

101.2.3. Keys

`@Id` identifies the property corresponding to the ID value.

You must annotate one of your POJO's fields as the ID value, because every entity in Cloud Datastore requires a single ID value:

```
@Entity(name = "trades")
public class Trade {
    @Id
    @Field(name = "trade_id")
    String tradeId;

    @Field(name = "trader_id")
    String traderId;

    String action;

    Double price;

    Double shares;

    String symbol;
}
```

Datastore can automatically allocate integer ID values. If a POJO instance with a `Long` ID property is written to Cloud Datastore with `null` as the ID value, then Spring Data Cloud Datastore will obtain a newly allocated ID value from Cloud Datastore and set that in the POJO for saving. Because primitive `long` ID properties cannot be `null` and default to `0`, keys will not be allocated.

101.2.4. Fields

All accessible properties on POJOs are automatically recognized as a Cloud Datastore field. Field naming is generated by the `PropertyNameFieldNamingStrategy` by default defined on the `DatastoreMappingContext` bean. The `@Field` annotation optionally provides a different field name than that of the property.

101.2.5. Supported Types

Spring Data Cloud Datastore supports the following types for regular fields and elements of collections:

Type	Stored as
`com.google.cloud.Timestamp`	com.google.cloud.datastore.TimestampValue
`com.google.cloud.datastore.Blob`	com.google.cloud.datastore.BlobValue
`com.google.cloud.datastore.LatLng`	com.google.cloud.datastore.LatLngValue
`java.lang.Boolean`, `boolean`	com.google.cloud.datastore.BooleanValue
`java.lang.Double`, `double`	com.google.cloud.datastore.DoubleValue
`java.lang.Long`, `long`	com.google.cloud.datastore.LongValue
`java.lang.Integer`, `int`	com.google.cloud.datastore.LongValue
`java.lang.String`	com.google.cloud.datastore.StringValue

Type	Stored as
`com.google.cloud.datastore.Entity`	com.google.cloud.datastore.EntityValue
`com.google.cloud.datastore.Key`	com.google.cloud.datastore.KeyValue
`byte[]`	com.google.cloud.datastore.BlobValue
Java `enum` values	com.google.cloud.datastore.StringValue

In addition, all types that can be converted to the ones listed in the table by `org.springframework.core.convert.support.DefaultConversionService` are supported.

101.2.6. Custom types

Custom converters can be used extending the type support for user defined types.

1. Converters need to implement the `org.springframework.core.convert.converter.Converter` interface in both directions.
2. The user defined type needs to be mapped to one of the basic types supported by Cloud Datastore.
3. An instance of both Converters (read and write) needs to be passed to the `DatastoreCustomConversions` constructor, which then has to be made available as a `@Bean` for `DatastoreCustomConversions`.

For example:

We would like to have a field of type `Album` on our `Singer` POJO and want it to be stored as a string property:

```
@Entity
public class Singer {

    @Id
    String singerId;

    String name;

    Album album;
}
```

Where Album is a simple class:

```
public class Album {
    String albumName;

    LocalDate date;
}
```

We have to define the two converters:

```
    //Converter to write custom Album type
    static final Converter<Album, String> ALBUM_STRING_CONVERTER =
            new Converter<Album, String>() {
                @Override
                public String convert(Album album) {
                    return album.getAlbumName() + " " +
album.getDate().format(DateTimeFormatter.ISO_DATE);
                }
            };

    //Converters to read custom Album type
    static final Converter<String, Album> STRING_ALBUM_CONVERTER =
            new Converter<String, Album>() {
                @Override
                public Album convert(String s) {
                    String[] parts = s.split(" ");
                    return new Album(parts[0], LocalDate.parse(parts[parts.length -
1], DateTimeFormatter.ISO_DATE));
                }
            };
```

That will be configured in our `@Configuration` file:

```
@Configuration
public class ConverterConfiguration {
    @Bean
    public DatastoreCustomConversions datastoreCustomConversions() {
        return new DatastoreCustomConversions(
                Arrays.asList(
                        ALBUM_STRING_CONVERTER,
                        STRING_ALBUM_CONVERTER));
    }
}
```

101.2.7. Collections and arrays

Arrays and collections (types that implement `java.util.Collection`) of supported types are supported. They are stored as `com.google.cloud.datastore.ListValue`. Elements are converted to Cloud Datastore supported types individually. `byte[]` is an exception, it is converted to `com.google.cloud.datastore.Blob`.

101.2.8. Custom Converter for collections

Users can provide converters from `List<?>` to the custom collection type. Only read converter is necessary, the Collection API is used on the write side to convert a collection to the internal list type.

Collection converters need to implement the `org.springframework.core.convert.converter.Converter` interface.

Example:

Let's improve the Singer class from the previous example. Instead of a field of type `Album`, we would like to have a field of type `Set<Album>`:

```
@Entity
public class Singer {

    @Id
    String singerId;

    String name;

    Set<Album> albums;
}
```

We have to define a read converter only:

```
static final Converter<List<?>, Set<?>> LIST_SET_CONVERTER =
            new Converter<List<?>, Set<?>>() {
                @Override
                public Set<?> convert(List<?> source) {
                    return Collections.unmodifiableSet(new HashSet<>(source));
                }
            };
```

And add it to the list of custom converters:

```
@Configuration
public class ConverterConfiguration {
    @Bean
    public DatastoreCustomConversions datastoreCustomConversions() {
        return new DatastoreCustomConversions(
                Arrays.asList(
                        LIST_SET_CONVERTER,
                        ALBUM_STRING_CONVERTER,
                        STRING_ALBUM_CONVERTER));
    }
}
```

101.2.9. Inheritance Hierarchies

Java entity types related by inheritance can be stored in the same Kind. When reading and querying entities using `DatastoreRepository` or `DatastoreTemplate` with a superclass as the type

parameter, you can receive instances of subclasses if you annotate the superclass and its subclasses with `DiscriminatorField` and `DiscriminatorValue`:

```
@Entity(name = "pets")
@DiscriminatorField(field = "pet_type")
abstract class Pet {
    @Id
    Long id;

    abstract String speak();
}

@DiscriminatorValue("cat")
class Cat extends Pet {
    @Override
    String speak() {
        return "meow";
    }
}

@DiscriminatorValue("dog")
class Dog extends Pet {
    @Override
    String speak() {
        return "woof";
    }
}

@DiscriminatorValue("pug")
class Pug extends Dog {
    @Override
    String speak() {
        return "woof woof";
    }
}
```

Instances of all 3 types are stored in the `pets` Kind. Because a single Kind is used, all classes in the hierarchy must share the same ID property and no two instances of any type in the hierarchy can share the same ID value.

Entity rows in Cloud Datastore store their respective types' `DiscriminatorValue` in a field specified by the root superclass's `DiscriminatorField` (`pet_type` in this case). Reads and queries using a given type parameter will match each entity with its specific type. For example, reading a `List<Pet>` will produce a list containing instances of all 3 types. However, reading a `List<Dog>` will produce a list containing only `Dog` and `Pug` instances. You can include the `pet_type` discrimination field in your Java entities, but its type must be convertible to a collection or array of `String`. Any value set in the discrimination field will be overwritten upon write to Cloud Datastore.

101.3. Relationships

There are three ways to represent relationships between entities that are described in this section:

- Embedded entities stored directly in the field of the containing entity
- `@Descendant` annotated properties for one-to-many relationships
- `@Reference` annotated properties for general relationships without hierarchy
- `@LazyReference` similar to `@Reference`, but the entities are lazy-loaded when the property is accessed. (Note that the keys of the children are retrieved when the parent entity is loaded.)

101.3.1. Embedded Entities

Fields whose types are also annotated with `@Entity` are converted to `EntityValue` and stored inside the parent entity.

Here is an example of Cloud Datastore entity containing an embedded entity in JSON:

```
{
  "name" : "Alexander",
  "age" : 47,
  "child" : {"name" : "Philip"  }
}
```

This corresponds to a simple pair of Java entities:

```
import org.springframework.cloud.gcp.data.datastore.core.mapping.Entity;
import org.springframework.data.annotation.Id;

@Entity("parents")
public class Parent {
  @Id
  String name;

  Child child;
}

@Entity
public class Child {
  String name;
}
```

`Child` entities are not stored in their own kind. They are stored in their entirety in the `child` field of the `parents` kind.

Multiple levels of embedded entities are supported.

Embedded entities don't need to have `@Id` field, it is only required for top level entities.

Example:

Entities can hold embedded entities that are their own type. We can store trees in Cloud Datastore using this feature:

```
import org.springframework.cloud.gcp.data.datastore.core.mapping.Embedded;
import org.springframework.cloud.gcp.data.datastore.core.mapping.Entity;
import org.springframework.data.annotation.Id;

@Entity
public class EmbeddableTreeNode {
  @Id
  long value;

  EmbeddableTreeNode left;

  EmbeddableTreeNode right;

  Map<String, Long> longValues;

  Map<String, List<Timestamp>> listTimestamps;

  public EmbeddableTreeNode(long value, EmbeddableTreeNode left, EmbeddableTreeNode
right) {
    this.value = value;
    this.left = left;
    this.right = right;
  }
}
```

Maps

Maps will be stored as embedded entities where the key values become the field names in the embedded entity. The value types in these maps can be any regularly supported property type, and the key values will be converted to String using the configured converters.

Also, a collection of entities can be embedded; it will be converted to `ListValue` on write.

Example:

Instead of a binary tree from the previous example, we would like to store a general tree (each node can have an arbitrary number of children) in Cloud Datastore. To do that, we need to create a field of type `List<EmbeddableTreeNode>`:

```
import org.springframework.cloud.gcp.data.datastore.core.mapping.Embedded;
import org.springframework.data.annotation.Id;

public class EmbeddableTreeNode {
  @Id
  long value;

  List<EmbeddableTreeNode> children;

  Map<String, EmbeddableTreeNode> siblingNodes;

  Map<String, Set<EmbeddableTreeNode>> subNodeGroups;

  public EmbeddableTreeNode(List<EmbeddableTreeNode> children) {
    this.children = children;
  }
}
```

Because Maps are stored as entities, they can further hold embedded entities:

- Singular embedded objects in the value can be stored in the values of embedded Maps.
- Collections of embedded objects in the value can also be stored as the values of embedded Maps.
- Maps in the value are further stored as embedded entities with the same rules applied recursively for their values.

101.3.2. Ancestor-Descendant Relationships

Parent-child relationships are supported via the `@Descendants` annotation.

Unlike embedded children, descendants are fully-formed entities residing in their own kinds. The parent entity does not have an extra field to hold the descendant entities. Instead, the relationship is captured in the descendants' keys, which refer to their parent entities:

```
import org.springframework.cloud.gcp.data.datastore.core.mapping.Descendants;
import org.springframework.cloud.gcp.data.datastore.core.mapping.Entity;
import org.springframework.data.annotation.Id;

@Entity("orders")
public class ShoppingOrder {
  @Id
  long id;

  @Descendants
  List<Item> items;
}

@Entity("purchased_item")
public class Item {
  @Id
  Key purchasedItemKey;

  String name;

  Timestamp timeAddedToOrder;
}
```

For example, an instance of a GQL key-literal representation for `Item` would also contain the parent `ShoppingOrder` ID value:

```
Key(orders, '12345', purchased_item, 'eggs')
```

The GQL key-literal representation for the parent `ShoppingOrder` would be:

```
Key(orders, '12345')
```

The Cloud Datastore entities exist separately in their own kinds.

The `ShoppingOrder`:

```
{
  "id" : 12345
}
```

The two items inside that order:

```
{
  "purchasedItemKey" : Key(orders, '12345', purchased_item, 'eggs'),
  "name" : "eggs",
  "timeAddedToOrder" : "2014-09-27 12:30:00.45-8:00"
}

{
  "purchasedItemKey" : Key(orders, '12345', purchased_item, 'sausage'),
  "name" : "sausage",
  "timeAddedToOrder" : "2014-09-28 11:30:00.45-9:00"
}
```

The parent-child relationship structure of objects is stored in Cloud Datastore using Datastore's ancestor relationships. Because the relationships are defined by the Ancestor mechanism, there is no extra column needed in either the parent or child entity to store this relationship. The relationship link is part of the descendant entity's key value. These relationships can be many levels deep.

Properties holding child entities must be collection-like, but they can be any of the supported inter-convertible collection-like types that are supported for regular properties such as `List`, arrays, `Set`, etc... Child items must have `Key` as their ID type because Cloud Datastore stores the ancestor relationship link inside the keys of the children.

Reading or saving an entity automatically causes all subsequent levels of children under that entity to be read or saved, respectively. If a new child is created and added to a property annotated `@Descendants` and the key property is left null, then a new key will be allocated for that child. The ordering of the retrieved children may not be the same as the ordering in the original property that was saved.

Child entities cannot be moved from the property of one parent to that of another unless the child's key property is set to `null` or a value that contains the new parent as an ancestor. Since Cloud Datastore entity keys can have multiple parents, it is possible that a child entity appears in the property of multiple parent entities. Because entity keys are immutable in Cloud Datastore, to change the key of a child you must delete the existing one and re-save it with the new key.

101.3.3. Key Reference Relationships

General relationships can be stored using the `@Reference` annotation.

```
import org.springframework.data.annotation.Reference;
import org.springframework.data.annotation.Id;

@Entity
public class ShoppingOrder {
  @Id
  long id;

  @Reference
  List<Item> items;

  @Reference
  Item specialSingleItem;
}

@Entity
public class Item {
  @Id
  Key purchasedItemKey;

  String name;

  Timestamp timeAddedToOrder;
}
```

`@Reference` relationships are between fully-formed entities residing in their own kinds. The relationship between `ShoppingOrder` and `Item` entities are stored as a Key field inside `ShoppingOrder`, which are resolved to the underlying Java entity type by Spring Data Cloud Datastore:

```
{
  "id" : 12345,
  "specialSingleItem" : Key(item, "milk"),
  "items" : [ Key(item, "eggs"), Key(item, "sausage") ]
}
```

Reference properties can either be singular or collection-like. These properties correspond to actual columns in the entity and Cloud Datastore Kind that hold the key values of the referenced entities. The referenced entities are full-fledged entities of other Kinds.

Similar to the `@Descendants` relationships, reading or writing an entity will recursively read or write all of the referenced entities at all levels. If referenced entities have `null` ID values, then they will be saved as new entities and will have ID values allocated by Cloud Datastore. There are no requirements for relationships between the key of an entity and the keys that entity holds as references. The order of collection-like reference properties is not preserved when reading back from Cloud Datastore.

101.4. Datastore Operations & Template

`DatastoreOperations` and its implementation, `DatastoreTemplate`, provides the Template pattern familiar to Spring developers.

Using the auto-configuration provided by Spring Boot Starter for Datastore, your Spring application context will contain a fully configured `DatastoreTemplate` object that you can autowire in your application:

```
@SpringBootApplication
public class DatastoreTemplateExample {

    @Autowired
    DatastoreTemplate datastoreTemplate;

    public void doSomething() {
        this.datastoreTemplate.deleteAll(Trader.class);
        //...
        Trader t = new Trader();
        //...
        this.datastoreTemplate.save(t);
        //...
        List<Trader> traders = datastoreTemplate.findAll(Trader.class);
        //...
    }
}
```

The Template API provides convenience methods for:

- Write operations (saving and deleting)
- Read-write transactions

101.4.1. GQL Query

In addition to retrieving entities by their IDs, you can also submit queries.

```
  <T> Iterable<T> query(Query<? extends BaseEntity> query, Class<T> entityClass);

  <A, T> Iterable<T> query(Query<A> query, Function<A, T> entityFunc);

  Iterable<Key> queryKeys(Query<Key> query);
```

These methods, respectively, allow querying for: * entities mapped by a given entity class using all the same mapping and converting features * arbitrary types produced by a given mapping function * only the Cloud Datastore keys of the entities found by the query

101.4.2. Find by ID(s)

Using `DatastoreTemplate` you can find entities by id. For example:

```
Trader trader = this.datastoreTemplate.findById("trader1", Trader.class);

List<Trader> traders = this.datastoreTemplate.findAllById(Arrays.asList("trader1",
"trader2"), Trader.class);

List<Trader> allTraders = this.datastoreTemplate.findAll(Trader.class);
```

Cloud Datastore uses key-based reads with strong consistency, but queries with eventual consistency. In the example above the first two reads utilize keys, while the third is run by using a query based on the corresponding Kind of `Trader`.

Indexes

By default, all fields are indexed. To disable indexing on a particular field, `@Unindexed` annotation can be used.

Example:

```
import org.springframework.cloud.gcp.data.datastore.core.mapping.Unindexed;

public class ExampleItem {
    long indexedField;

    @Unindexed
    long unindexedField;

    @Unindexed
    List<String> unindexedListField;
}
```

When using queries directly or via Query Methods, Cloud Datastore requires composite custom indexes if the select statement is not `SELECT *` or if there is more than one filtering condition in the `WHERE` clause.

Read with offsets, limits, and sorting

`DatastoreRepository` and custom-defined entity repositories implement the Spring Data `PagingAndSortingRepository`, which supports offsets and limits using page numbers and page sizes. Paging and sorting options are also supported in `DatastoreTemplate` by supplying a `DatastoreQueryOptions` to `findAll`.

Partial read

This feature is not supported yet.

101.4.3. Write / Update

The write methods of `DatastoreOperations` accept a POJO and writes all of its properties to Datastore. The required Datastore kind and entity metadata is obtained from the given object's actual type.

If a POJO was retrieved from Datastore and its ID value was changed and then written or updated, the operation will occur as if against a row with the new ID value. The entity with the original ID value will not be affected.

```
Trader t = new Trader();
this.datastoreTemplate.save(t);
```

The `save` method behaves as update-or-insert.

Partial Update

This feature is not supported yet.

101.4.4. Transactions

Read and write transactions are provided by `DatastoreOperations` via the `performTransaction` method:

```
@Autowired
DatastoreOperations myDatastoreOperations;

public String doWorkInsideTransaction() {
  return myDatastoreOperations.performTransaction(
    transactionDatastoreOperations -> {
      // Work with transactionDatastoreOperations here.
      // It is also a DatastoreOperations object.

      return "transaction completed";
    }
  );
}
```

The `performTransaction` method accepts a `Function` that is provided an instance of a `DatastoreOperations` object. The final returned value and type of the function is determined by the user. You can use this object just as you would a regular `DatastoreOperations` with an exception:

- It cannot perform sub-transactions.

Because of Cloud Datastore's consistency guarantees, there are limitations to the operations and relationships among entities used inside transactions.

Declarative Transactions with @Transactional Annotation

This feature requires a bean of `DatastoreTransactionManager`, which is provided when using `spring-cloud-gcp-starter-data-datastore`.

`DatastoreTemplate` and `DatastoreRepository` support running methods with the `@Transactional` annotation as transactions. If a method annotated with `@Transactional` calls another method also annotated, then both methods will work within the same transaction. `performTransaction` cannot be used in `@Transactional` annotated methods because Cloud Datastore does not support transactions within transactions.

101.4.5. Read-Write Support for Maps

You can work with Maps of type `Map<String, ?>` instead of with entity objects by directly reading and writing them to and from Cloud Datastore.

This is a different situation than using entity objects that contain Map properties.

The map keys are used as field names for a Datastore entity and map values are converted to Datastore supported types. Only simple types are supported (i.e. collections are not supported). Converters for custom value types can be added (see Custom types section).

Example:

```
Map<String, Long> map = new HashMap<>();
map.put("field1", 1L);
map.put("field2", 2L);
map.put("field3", 3L);

keyForMap = datastoreTemplate.createKey("kindName", "id");

//write a map
datastoreTemplate.writeMap(keyForMap, map);

//read a map
Map<String, Long> loadedMap = datastoreTemplate.findByIdAsMap(keyForMap, Long.class);
```

101.5. Repositories

Spring Data Repositories are an abstraction that can reduce boilerplate code.

For example:

```
public interface TraderRepository extends DatastoreRepository<Trader, String> {
}
```

Spring Data generates a working implementation of the specified interface, which can be autowired

into an application.

The `Trader` type parameter to `DatastoreRepository` refers to the underlying domain type. The second type parameter, `String` in this case, refers to the type of the key of the domain type.

```
public class MyApplication {

    @Autowired
    TraderRepository traderRepository;

    public void demo() {

        this.traderRepository.deleteAll();
        String traderId = "demo_trader";
        Trader t = new Trader();
        t.traderId = traderId;
        this.tradeRepository.save(t);

        Iterable<Trader> allTraders = this.traderRepository.findAll();

        int count = this.traderRepository.count();
    }
}
```

Repositories allow you to define custom Query Methods (detailed in the following sections) for retrieving, counting, and deleting based on filtering and paging parameters. Filtering parameters can be of types supported by your configured custom converters.

101.5.1. Query methods by convention

```
public interface TradeRepository extends DatastoreRepository<Trade, String[]> {
  List<Trader> findByAction(String action);

  //throws an exception if no results
  Trader findOneByAction(String action);

  //because of the annotation, returns null if no results
  @Nullable
  Trader getByAction(String action);

  Optional<Trader> getOneByAction(String action);

  int countByAction(String action);

  boolean existsByAction(String action);

  List<Trade>
findTop3ByActionAndSymbolAndPriceGreaterThanAndPriceLessThanOrEqualOrderBySymbolDesc(
            String action, String symbol, double priceFloor, double priceCeiling);

  Page<TestEntity> findByAction(String action, Pageable pageable);

  Slice<TestEntity> findBySymbol(String symbol, Pageable pageable);

  List<TestEntity> findBySymbol(String symbol, Sort sort);
}
```

In the example above the query methods in `TradeRepository` are generated based on the name of the methods using the Spring Data Query creation naming convention.

You can refer to nested fields using Spring Data JPA Property Expressions

Cloud Datastore only supports filter components joined by AND, and the following operations:

- `equals`
- `greater than or equals`
- `greater than`
- `less than or equals`
- `less than`
- `is null`

After writing a custom repository interface specifying just the signatures of these methods, implementations are generated for you and can be used with an auto-wired instance of the repository. Because of Cloud Datastore's requirement that explicitly selected fields must all appear in a composite index together, `find` name-based query methods are run as `SELECT *`.

Delete queries are also supported. For example, query methods such as `deleteByAction` or `removeByAction` delete entities found by `findByAction`. Delete queries are run as separate read and

delete operations instead of as a single transaction because Cloud Datastore cannot query in transactions unless ancestors for queries are specified. As a result, `removeBy` and `deleteBy` name-convention query methods cannot be used inside transactions via either `performInTransaction` or `@Transactional` annotation.

Delete queries can have the following return types:

- An integer type that is the number of entities deleted
- A collection of entities that were deleted
- 'void'

Methods can have `org.springframework.data.domain.Pageable` parameter to control pagination and sorting, or `org.springframework.data.domain.Sort` parameter to control sorting only. See Spring Data documentation for details.

For returning multiple items in a repository method, we support Java collections as well as `org.springframework.data.domain.Page` and `org.springframework.data.domain.Slice`. If a method's return type is `org.springframework.data.domain.Page`, the returned object will include current page, total number of results and total number of pages.

Methods that return `Page` run an additional query to compute total number of pages. Methods that return `Slice`, on the other hand, do not run any additional queries and, therefore, are much more efficient.

101.5.2. Empty result handling in repository methods

Java `java.util.Optional` can be used to indicate the potential absence of a return value.

Alternatively, query methods can return the result without a wrapper. In that case the absence of a query result is indicated by returning `null`. Repository methods returning collections are guaranteed never to return `null` but rather the corresponding empty collection.

You can enable nullability checks. For more details please see Spring Framework's nullability docs.

101.5.3. Query by example

Query by Example is an alternative querying technique. It enables dynamic query generation based on a user-provided object. See Spring Data Documentation for details.

Unsupported features:

1. Currently, only equality queries are supported (no ignore-case matching, regexp matching, etc.).
2. Per-field matchers are not supported.
3. Embedded entities matching is not supported.

For example, if you want to find all users with the last name "Smith", you would use the following code:

```
userRepository.findAll(
    Example.of(new User(null, null, "Smith"))
```

`null` fields are not used in the filter by default. If you want to include them, you would use the following code:

```
userRepository.findAll(
    Example.of(new User(null, null, "Smith"),
ExampleMatcher.matching().withIncludeNullValues())
```

101.5.4. Custom GQL query methods

Custom GQL queries can be mapped to repository methods in one of two ways:

- `namedQueries` properties file
- using the `@Query` annotation

Query methods with annotation

Using the `@Query` annotation:

The names of the tags of the GQL correspond to the `@Param` annotated names of the method parameters.

```
public interface TraderRepository extends DatastoreRepository<Trader, String> {

  @Query("SELECT * FROM traders WHERE name = @trader_name")
  List<Trader> tradersByName(@Param("trader_name") String traderName);

  @Query("SELECT * FROM  test_entities_ci WHERE name = @trader_name")
  TestEntity getOneTestEntity(@Param("trader_name") String traderName);

  @Query("SELECT * FROM traders WHERE name = @trader_name")
  List<Trader> tradersByNameSort(@Param("trader_name") String traderName, Sort sort);

  @Query("SELECT * FROM traders WHERE name = @trader_name")
  Slice<Trader> tradersByNameSlice(@Param("trader_name") String traderName, Pageable
pageable);

  @Query("SELECT * FROM traders WHERE name = @trader_name")
  Page<Trader> tradersByNamePage(@Param("trader_name") String traderName, Pageable
pageable);
}
```

When the return type is `Slice` or `Pageable`, the result set cursor that points to the position just after the page is preserved in the returned `Slice` or `Page` object. To take advantage of the cursor to query

for the next page or slice, use `result.getPageable().next()`.

`Page` requires the total count of entities produced by the query. Therefore, the first query will have to retrieve all of the records just to count them. Instead, we recommend using the `Slice` return type, because it does not require an additional count query.

```
Slice<Trader> slice1 = tradersByNamePage("Dave", PageRequest.of(0, 5));
Slice<Trader> slice2 = tradersByNamePage("Dave", slice1.getPageable().next());
```

You cannot use these Query Methods in repositories where the type parameter is a subclass of another class annotated with `DiscriminatorField`.

The following parameter types are supported:

- `com.google.cloud.Timestamp`
- `com.google.cloud.datastore.Blob`
- `com.google.cloud.datastore.Key`
- `com.google.cloud.datastore.Cursor`
- `java.lang.Boolean`
- `java.lang.Double`
- `java.lang.Long`
- `java.lang.String`
- `enum` values. These are queried as `String` values.

With the exception of `Cursor`, array forms of each of the types are also supported.

If you would like to obtain the count of items of a query or if there are any items returned by the query, set the `count = true` or `exists = true` properties of the `@Query` annotation, respectively. The return type of the query method in these cases should be an integer type or a boolean type.

Cloud Datastore provides provides the `SELECT __key__ FROM …` special column for all kinds that retrieves the `Key` of each row. Selecting this special `__key__` column is especially useful and efficient for `count` and `exists` queries.

You can also query for non-entity types:

```
@Query(value = "SELECT __key__ from test_entities_ci")
List<Key> getKeys();

@Query(value = "SELECT __key__ from test_entities_ci limit 1")
Key getKey();
```

In order to use `@Id` annotated fields in custom queries, use `__key__` keyword for the field name. The parameter type should be of `Key`, as in the following example.

Repository method:

```
@Query("select * from  test_entities_ci where size = @size and __key__ = @id")
LinkedList<TestEntity> findEntities(@Param("size") long size, @Param("id") Key id);
```

Generate a key from id value using `DatastoreTemplate.createKey` method and use it as a parameter for the repository method:

```
this.testEntityRepository.findEntities(1L,
datastoreTemplate.createKey(TestEntity.class, 1L))
```

SpEL can be used to provide GQL parameters:

```
@Query("SELECT * FROM |com.example.Trade| WHERE trades.action = @act
  AND price > :#{#priceRadius * -1} AND price < :#{#priceRadius * 2}")
List<Trade> fetchByActionNamedQuery(@Param("act") String action, @Param("priceRadius")
Double r);
```

Kind names can be directly written in the GQL annotations. Kind names can also be resolved from the `@Entity` annotation on domain classes.

In this case, the query should refer to table names with fully qualified class names surrounded by `|` characters: `|fully.qualified.ClassName|`. This is useful when SpEL expressions appear in the kind name provided to the `@Entity` annotation. For example:

```
@Query("SELECT * FROM |com.example.Trade| WHERE trades.action = @act")
List<Trade> fetchByActionNamedQuery(@Param("act") String action);
```

Query methods with named queries properties

You can also specify queries with Cloud Datastore parameter tags and SpEL expressions in properties files.

By default, the `namedQueriesLocation` attribute on `@EnableDatastoreRepositories` points to the `META-INF/datastore-named-queries.properties` file. You can specify the query for a method in the properties file by providing the GQL as the value for the "interface.method" property:

You cannot use these Query Methods in repositories where the type parameter is a subclass of another class annotated with `DiscriminatorField`.

```
Trader.fetchByName=SELECT * FROM traders WHERE name = @tag0
```

```
public interface TraderRepository extends DatastoreRepository<Trader, String> {

    // This method uses the query from the properties file instead of one generated
based on name.
    List<Trader> fetchByName(@Param("tag0") String traderName);

}
```

101.5.5. Transactions

These transactions work very similarly to those of `DatastoreOperations`, but is specific to the repository's domain type and provides repository functions instead of template functions.

For example, this is a read-write transaction:

```
@Autowired
DatastoreRepository myRepo;

public String doWorkInsideTransaction() {
  return myRepo.performTransaction(
    transactionDatastoreRepo -> {
      // Work with the single-transaction transactionDatastoreRepo here.
      // This is a DatastoreRepository object.

      return "transaction completed";
    }
  );
}
```

101.5.6. Projections

Spring Data Cloud Datastore supports projections. You can define projection interfaces based on domain types and add query methods that return them in your repository:

```
public interface TradeProjection {

    String getAction();

    @Value("#{target.symbol + ' ' + target.action}")
    String getSymbolAndAction();
}

public interface TradeRepository extends DatastoreRepository<Trade, Key> {

    List<Trade> findByTraderId(String traderId);

    List<TradeProjection> findByAction(String action);

    @Query("SELECT action, symbol FROM trades WHERE action = @action")
    List<TradeProjection> findByQuery(String action);
}
```

Projections can be provided by name-convention-based query methods as well as by custom GQL queries. If using custom GQL queries, you can further restrict the fields retrieved from Cloud Datastore to just those required by the projection. However, custom select statements (those not using `SELECT *`) require composite indexes containing the selected fields.

Properties of projection types defined using SpEL use the fixed name `target` for the underlying domain object. As a result, accessing underlying properties take the form `target.<property-name>`.

101.5.7. REST Repositories

When running with Spring Boot, repositories can be exposed as REST services by simply adding this dependency to your pom file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

If you prefer to configure parameters (such as path), you can use `@RepositoryRestResource` annotation:

```
@RepositoryRestResource(collectionResourceRel = "trades", path = "trades")
public interface TradeRepository extends DatastoreRepository<Trade, String[]> {
}
```

For example, you can retrieve all `Trade` objects in the repository by using `curl http://<server>:<port>/trades`, or any specific trade via `curl http://<server>:<port>/trades/<trader_id>`.

You can also write trades using `curl -XPOST -H"Content-Type: application/json" -d@test.json http://<server>:<port>/trades/` where the file `test.json` holds the JSON representation of a `Trade` object.

To delete trades, you can use `curl -XDELETE http://<server>:<port>/trades/<trader_id>`

101.6. Events

Spring Data Cloud Datastore publishes events extending the Spring Framework's `ApplicationEvent` to the context that can be received by `ApplicationListener` beans you register.

Type	Description	Contents
`AfterFindByKeyEvent`	Published immediately after read by-key operations are run by `DatastoreTemplate`	The entities read from Cloud Datastore and the original keys in the request.
`AfterQueryEvent`	Published immediately after read byquery operations are run by `DatastoreTemplate`	The entities read from Cloud Datastore and the original query in the request.
`BeforeSaveEvent`	Published immediately before save operations are run by `DatastoreTemplate`	The entities to be sent to Cloud Datastore and the original Java objects being saved.
`AfterSaveEvent`	Published immediately after save operations are run by `DatastoreTemplate`	The entities sent to Cloud Datastore and the original Java objects being saved.
`BeforeDeleteEvent`	Published immediately before delete operations are run by `DatastoreTemplate`	The keys to be sent to Cloud Datastore. The target entities, ID values, or entity type originally specified for the delete operation.
`AfterDeleteEvent`	Published immediately after delete operations are run by `DatastoreTemplate`	The keys sent to Cloud Datastore. The target entities, ID values, or entity type originally specified for the delete operation.

101.7. Auditing

Spring Data Cloud Datastore supports the `@LastModifiedDate` and `@LastModifiedBy` auditing annotations for properties:

```
@Entity
public class SimpleEntity {
    @Id
    String id;

    @LastModifiedBy
    String lastUser;

    @LastModifiedDate
    DateTime lastTouched;
}
```

Upon insert, update, or save, these properties will be set automatically by the framework before Datastore entities are generated and saved to Cloud Datastore.

To take advantage of these features, add the `@EnableDatastoreAuditing` annotation to your configuration class and provide a bean for an `AuditorAware<A>` implementation where the type `A` is the desired property type annotated by `@LastModifiedBy`:

```
@Configuration
@EnableDatastoreAuditing
public class Config {

    @Bean
    public AuditorAware<String> auditorProvider() {
        return () -> Optional.of("YOUR_USERNAME_HERE");
    }
}
```

The `AuditorAware` interface contains a single method that supplies the value for fields annotated by `@LastModifiedBy` and can be of any type. One alternative is to use Spring Security's `User` type:

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  public Optional<User> getCurrentAuditor() {

    return Optional.ofNullable(SecurityContextHolder.getContext())
              .map(SecurityContext::getAuthentication)
              .filter(Authentication::isAuthenticated)
              .map(Authentication::getPrincipal)
              .map(User.class::cast);
  }
}
```

You can also set a custom provider for properties annotated `@LastModifiedDate` by providing a bean for `DateTimeProvider` and providing the bean name to `@EnableDatastoreAuditing(dateTimeProviderRef = "customDateTimeProviderBean")`.

101.8. Partitioning Data by Namespace

You can partition your data by using more than one namespace. This is the recommended method for multi-tenancy in Cloud Datastore.

```
    @Bean
    public DatastoreNamespaceProvider namespaceProvider() {
        // return custom Supplier of a namespace string.
    }
```

The `DatastoreNamespaceProvider` is a synonym for `Supplier<String>`. By providing a custom implementation of this bean (for example, supplying a thread-local namespace name), you can direct your application to use multiple namespaces. Every read, write, query, and transaction you perform will utilize the namespace provided by this supplier.

Note that your provided namespace in `application.properties` will be ignored if you define a namespace provider bean.

101.9. Spring Boot Actuator Support

101.9.1. Cloud Datastore Health Indicator

If you are using Spring Boot Actuator, you can take advantage of the Cloud Datastore health indicator called `datastore`. The health indicator will verify whether Cloud Datastore is up and accessible by your application. To enable it, all you need to do is add the Spring Boot Actuator to your project.

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

101.10. Sample

A Simple Spring Boot Application and more advanced Sample Spring Boot Application are provided to show how to use the Spring Data Cloud Datastore starter and template.

Chapter 102. Spring Data Cloud Firestore

Currently some features are not supported: query by example, projections, and auditing.

Spring Data is an abstraction for storing and retrieving POJOs in numerous storage technologies. Spring Cloud GCP adds Spring Data Reactive Repositories support for Google Cloud Firestore in native mode, providing reactive template and repositories support. To begin using this library, add the `spring-cloud-gcp-data-firestore` artifact to your project.

Maven coordinates for this module only, using Spring Cloud GCP BOM:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-data-firestore</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-data-firestore'
}
```

We provide a Spring Boot Starter for Spring Data Firestore, with which you can use our recommended auto-configuration setup. To use the starter, see the coordinates below.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-data-firestore</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-data-
firestore'
}
```

102.1. Configuration

102.1.1. Properties

The Spring Boot starter for Google Cloud Firestore provides the following configuration options:

Name	Description	Required	Default value
`spring.cloud.gcp.firestore.enabled`	Enables or disables Firestore auto-configuration	No	`true`
`spring.cloud.gcp.firestore.project-id`	GCP project ID where the Google Cloud Firestore API is hosted, if different from the one in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.firestore.emulator.enabled`	Enables the usage of an emulator. If this is set to true, then you should set the `spring.cloud.gcp.firestore.host-port` to the host:port of your locally running emulator instance	No	`false`
`spring.cloud.gcp.firestore.host-port`	The host and port of the Firestore service; can be overridden to specify connecting to an already-running Firestore emulator instance.	No	`firestore.googleapis.com:443` (the host/port of official Firestore service)
`spring.cloud.gcp.firestore.credentials.location`	OAuth2 credentials for authenticating with the Google Cloud Firestore API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.firestore.credentials.encoded-key`	Base64-encoded OAuth2 credentials for authenticating with the Google Cloud Firestore API, if different from the ones in the Spring Cloud GCP Core Module	No	
`spring.cloud.gcp.firestore.credentials.scopes`	OAuth2 scope for Spring Cloud GCP Cloud Firestore credentials	No	www.googleapis.com/auth/datastore

102.1.2. Supported types

You may use the following field types when defining your persistent entities or when binding query parameters:

- `Long`
- `Integer`
- `Double`
- `Float`
- `String`
- `Boolean`
- `Character`
- `Date`
- `Map`
- `List`
- `Enum`
- `com.google.cloud.Timestamp`
- `com.google.cloud.firestore.GeoPoint`
- `com.google.cloud.firestore.Blob`

102.1.3. Reactive Repository settings

Spring Data Repositories can be configured via the `@EnableReactiveFirestoreRepositories` annotation on your main `@Configuration` class. With our Spring Boot Starter for Spring Data Cloud Firestore, `@EnableReactiveFirestoreRepositories` is automatically added. It is not required to add it to any other class, unless there is a need to override finer grain configuration parameters provided by `@EnableReactiveFirestoreRepositories`.

102.1.4. Autoconfiguration

Our Spring Boot autoconfiguration creates the following beans available in the Spring application context:

- an instance of `FirestoreTemplate`
- instances of all user defined repositories extending `FirestoreReactiveRepository` (an extension of `ReactiveCrudRepository` with additional Cloud Firestore features) when repositories are enabled
- an instance of `Firestore` from the Google Cloud Java Client for Firestore, for convenience and lower level API access

102.2. Object Mapping

Spring Data Cloud Firestore allows you to map domain POJOs to Cloud Firestore collections and documents via annotations:

```
import com.google.cloud.firestore.annotation.DocumentId;
import org.springframework.cloud.gcp.data.firestore.Document;

@Document(collectionName = "usersCollection")
public class User {
    /**
     * Used to test @PropertyName annotation on a field.
     */
    @PropertyName("drink")
    public String favoriteDrink;

    @DocumentId
    private String name;

    private Integer age;

    public User() {
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return this.age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

`@Document(collectionName = "usersCollection")` annotation configures the collection name for the documents of this type. This annotation is optional, by default the collection name is derived from the class name.

`@DocumentId` annotation marks a field to be used as document id. This annotation is required and the annotated field can only be of `String` type.

If the property annotated with `@DocumentId` is `null` the document id is generated automatically when the entity is saved.

Internally we use Firestore client library object mapping. See the documentation for supported annotations.

102.2.1. Embedded entities and lists

Spring Data Cloud Firestore supports embedded properties of custom types and lists. Given a custom POJO definition, you can have properties of this type or lists of this type in your entities. They are stored as embedded documents (or arrays, correspondingly) in the Cloud Firestore.

Example:

```
@Document(collectionName = "usersCollection")
public class User {
    /**
     * Used to test @PropertyName annotation on a field.
     */
    @PropertyName("drink")
    public String favoriteDrink;

    @DocumentId
    private String name;

    private Integer age;

    private List<String> pets;

    private List<Address> addresses;

    private Address homeAddress;

    public List<String> getPets() {
        return this.pets;
    }

    public void setPets(List<String> pets) {
        this.pets = pets;
    }

    public List<Address> getAddresses() {
        return this.addresses;
    }

    public void setAddresses(List<Address> addresses) {
        this.addresses = addresses;
    }

    @PropertyName("address")
    public Address getHomeAddress() {
        return this.homeAddress;
    }

    @PropertyName("address")
    public void setHomeAddress(Address homeAddress) {
```

```
        this.homeAddress = homeAddress;
    }

    public static class Address {
        String streetAddress;
        String country;

        public Address() {
        }
    }
}
```

102.3. Reactive Repositories

Spring Data Repositories is an abstraction that can reduce boilerplate code.

For example:

```
public interface UserRepository extends FirestoreReactiveRepository<User> {
    Flux<User> findByAge(Integer age);

    Flux<User> findByHomeAddressCountry(String country);

    Flux<User> findByFavoriteDrink(String drink);

    Flux<User> findByAgeGreaterThanAndAgeLessThan(Integer age1, Integer age2);

    Flux<User> findByAgeGreaterThan(Integer age);

    Flux<User> findByAgeGreaterThan(Integer age, Sort sort);

    Flux<User> findByAgeGreaterThan(Integer age, Pageable pageable);

    Flux<User> findByAgeIn(List<Integer> ages);

    Flux<User> findByAgeAndPetsContains(Integer age, List<String> pets);

    Flux<User> findByPetsContains(List<String> pets);

    Flux<User> findByPetsContainsAndAgeIn(String pets, List<Integer> ages);

    Mono<Long> countByAgeIsGreaterThan(Integer age);
}
```

Spring Data generates a working implementation of the specified interface, which can be autowired into an application.

The `User` type parameter to `FirestoreReactiveRepository` refers to the underlying domain type.

You can refer to nested fields using Spring Data JPA Property Expressions

```
public class MyApplication {

    @Autowired
    UserRepository userRepository;

    public void writeReadDeleteTest() {
        List<User.Address> addresses = Arrays.asList(new User.Address("123 Alice st",
"US"),
                new User.Address("1 Alice ave", "US"));
        User.Address homeAddress = new User.Address("10 Alice blvd", "UK");
        User alice = new User("Alice", 29, null, addresses, homeAddress);
        User bob = new User("Bob", 60);

        this.userRepository.save(alice).block();
        this.userRepository.save(bob).block();

        assertThat(this.userRepository.count().block()).isEqualTo(2);

assertThat(this.userRepository.findAll().map(User::getName).collectList().block())
                .containsExactlyInAnyOrder("Alice", "Bob");

        User aliceLoaded = this.userRepository.findById("Alice").block();
        assertThat(aliceLoaded.getAddresses()).isEqualTo(addresses);
        assertThat(aliceLoaded.getHomeAddress()).isEqualTo(homeAddress);
    }
}
```

Repositories allow you to define custom Query Methods (detailed in the following sections) for retrieving and counting based on filtering and paging parameters.

Custom queries with `@Query` annotation are not supported since there is no query language in Cloud Firestore

102.4. Firestore Operations & Template

`FirestoreOperations` and its implementation, `FirestoreTemplate`, provides the Template pattern familiar to Spring developers.

Using the auto-configuration provided by Spring Data Cloud Firestore, your Spring application context will contain a fully configured `FirestoreTemplate` object that you can autowire in your application:

```
@SpringBootApplication
public class FirestoreTemplateExample {

    @Autowired
    FirestoreOperations firestoreOperations;

    public Mono<User> createUsers() {
        return this.firestoreOperatons.save(new User("Alice", 29))
            .then(this.firestoreOperatons.save(new User("Bob", 60)));
    }

    public Flux<User> findUsers() {
        return this.firestoreOperatons.findAll(User.class);
    }

    public Mono<Long> removeAllUsers() {
        return this.firestoreOperatons.deleteAll(User.class);
    }
}
```

The Template API provides support for:

- Read and write operations
- Transactions
- Subcollections operations

102.5. Query methods by convention

```
public class MyApplication {
    public void partTreeRepositoryMethodTest() {
        User u1 = new User("Cloud", 22, null, null, new Address("1 First st., NYC",
"USA"));
        u1.favoriteDrink = "tea";
        User u2 = new User("Squall", 17, null, null, new Address("2 Second st.,
London", "UK"));
        u2.favoriteDrink = "wine";
        Flux<User> users = Flux.fromArray(new User[] {u1, u2});

        this.userRepository.saveAll(users).blockLast();

        assertThat(this.userRepository.count().block()).isEqualTo(2);

assertThat(this.userRepository.findByAge(22).collectList().block()).containsExactly(u1
);

assertThat(this.userRepository.findByHomeAddressCountry("USA").collectList().block()).
containsExactly(u1);

assertThat(this.userRepository.findByFavoriteDrink("wine").collectList().block()).cont
ainsExactly(u2);
        assertThat(this.userRepository.findByAgeGreaterThanAndAgeLessThan(20,
30).collectList().block())
                .containsExactly(u1);

assertThat(this.userRepository.findByAgeGreaterThan(10).collectList().block()).contain
sExactlyInAnyOrder(u1,
                u2);
    }
}
```

In the example above the query method implementations in `UserRepository` are generated based on the name of the methods using the Spring Data Query creation naming convention.

Cloud Firestore only supports filter components joined by AND, and the following operations:

- `equals`
- `greater than or equals`
- `greater than`
- `less than or equals`
- `less than`
- `is null`
- `contains` (accepts a `List` with up to 10 elements, or a singular value)
- `in` (accepts a `List` with up to 10 elements)

If `in` operation is used in combination with `contains` operation, the argument to `contains` operation has to be a singular value.

After writing a custom repository interface specifying just the signatures of these methods, implementations are generated for you and can be used with an auto-wired instance of the repository.

102.6. Transactions

Read-only and read-write transactions are provided by `TransactionalOperator` (see this blog post on reactive transactions for details). In order to use it, you would need to autowire `ReactiveFirestoreTransactionManager` like this:

```
public class MyApplication {
    @Autowired
    ReactiveFirestoreTransactionManager txManager;
}
```

After that you will be able to use it to create an instance of `TransactionalOperator`. Note that you can switch between read-only and read-write transactions using `TransactionDefinition` object:

```
DefaultTransactionDefinition transactionDefinition = new
DefaultTransactionDefinition();
transactionDefinition.setReadOnly(false);
TransactionalOperator operator = TransactionalOperator.create(this.txManager,
transactionDefinition);
```

When you have an instance of `TransactionalOperator`, you can invoke a sequence of Firestore operations in a transaction by using `operator::transactional`:

```
User alice = new User("Alice", 29);
User bob = new User("Bob", 60);

this.userRepository.save(alice)
        .then(this.userRepository.save(bob))
        .as(operator::transactional)
        .block();

this.userRepository.findAll()
        .flatMap(a -> {
            a.setAge(a.getAge() - 1);
            return this.userRepository.save(a);
        })
        .as(operator::transactional).collectList().block();

assertThat(this.userRepository.findAll().map(User::getAge).collectList().block())
        .containsExactlyInAnyOrder(28, 59);
```

Read operations in a transaction can only happen before write operations. All write operations are applied atomically. Read documents are locked until the transaction finishes with a commit or a rollback, which are handled by Spring Data. If an `Exception` is thrown within a transaction, the rollback operation is performed. Otherwise, the commit operation is performed.

102.6.1. Declarative Transactions with @Transactional Annotation

This feature requires a bean of `SpannerTransactionManager`, which is provided when using `spring-cloud-gcp-starter-data-firestore`.

`FirestoreTemplate` and `FirestoreReactiveRepository` support running methods with the `@Transactional` annotation as transactions. If a method annotated with `@Transactional` calls another method also annotated, then both methods will work within the same transaction.

One way to use this feature is illustrated here. You would need to do the following:

1. Annotate your configuration class with the `@EnableTransactionManagement` annotation.
2. Create a service class that has methods annotated with `@Transactional`:

```
class UserService {
    @Autowired
    private UserRepository userRepository;

    @Transactional
    public Mono<Void> updateUsers() {
        return this.userRepository.findAll()
                .flatMap(a -> {
                    a.setAge(a.getAge() - 1);
                    return this.userRepository.save(a);
                })
                .then();
    }
}
```

3. Make a Spring Bean provider that creates an instance of that class:

```
@Bean
public UserService userService() {
    return new UserService();
}
```

After that, you can autowire your service like so:

```
public class MyApplication {
    @Autowired
    UserService userService;
}
```

Now when you call the methods annotated with `@Transactional` on your service object, a transaction will be automatically started. If an error occurs during the execution of a method annotated with `@Transactional`, the transaction will be rolled back. If no error occurs, the transaction will be committed.

102.6.2. Subcollections

A subcollection is a collection associated with a specific entity. Documents in subcollections can contain subcollections as well, allowing you to further nest data. You can nest data up to 100 levels deep.

Deleting a document does not delete its subcollections!

To use subcollections you will need to create a `FirestoreReactiveOperations` object with a parent entity using `FirestoreReactiveOperations.withParent` call. You can use this object to save, query and remove entities associated with this parent. The parent doesn't have to exist in Firestore, but should have a non-empty id field.

Autowire `FirestoreReactiveOperations`:

```
@Autowired
FirestoreReactiveOperations firestoreTemplate;
```

Then you can use this object to create a `FirestoreReactiveOperations` object with a custom parent:

```
FirestoreReactiveOperations bobTemplate = this.firestoreTemplate.withParent(new
User("Bob", 60));

PhoneNumber phoneNumber = new PhoneNumber("111-222-333");
bobTemplate.save(phoneNumber).block();
assertThat(bobTemplate.findAll(PhoneNumber.class).collectList().block()).containsExact
ly(phoneNumber);
bobTemplate.deleteAll(PhoneNumber.class).block();
assertThat(bobTemplate.findAll(PhoneNumber.class).collectList().block()).isEmpty();
```

102.7. Cloud Firestore Spring Boot Starter

If you prefer using Firestore client only, Spring Cloud GCP provides a convenience starter which automatically configures authentication settings and client objects needed to begin using Google Cloud Firestore in native mode.

See documentation to learn more about Cloud Firestore.

To begin using this library, add the `spring-cloud-gcp-starter-firestore` artifact to your project.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-firestore</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
firestore'
}
```

102.7.1. Using Cloud Firestore

The starter automatically configures and registers a `Firestore` bean in the Spring application context. To start using it, simply use the `@Autowired` annotation.

```
@Autowired
Firestore firestore;

   void writeDocumentFromObject() throws ExecutionException, InterruptedException {
       // Add document data with id "joe" using a custom User class
       User data = new User("Joe",
               Arrays.asList(
                       new Phone(12345, PhoneType.CELL),
                       new Phone(54321, PhoneType.WORK)));

       // .get() blocks on response
       WriteResult writeResult = this.firestore.document("users/joe").set(data).get();

       LOGGER.info("Update time: " + writeResult.getUpdateTime());
   }

   User readDocumentToObject() throws ExecutionException, InterruptedException {
           ApiFuture<DocumentSnapshot> documentFuture =
                   this.firestore.document("users/joe").get();

           User user = documentFuture.get().toObject(User.class);

           LOGGER.info("read: " + user);

           return user;
   }
```

102.8. Emulator Usage

The Google Cloud Firebase SDK provides a local, in-memory emulator for Cloud Firestore, which you can use to develop and test your application.

First follow the Firebase emulator installation steps to install, configure, and run the emulator.

By default, the emulator is configured to run on port 8080; you will need to ensure that the emulator does not run on the same port as your Spring application.

Once the Firestore emulator is running, ensure that the following properties are set in your `application.properties` of your Spring application:

```
spring.cloud.gcp.firestore.emulator.enabled=true
spring.cloud.gcp.firestore.host-port=${EMULATOR_HOSTPORT}
```

From this point onward, your application will connect to your locally running emulator instance instead of the real Firestore service.

102.9. Samples

Spring Cloud GCP provides Firestore sample applications to demonstrate API usage:

- Reactive Firestore Repository sample application:
- Firestore Client Library sample application

Chapter 103. Cloud Memorystore for Redis

103.1. Spring Caching

Cloud Memorystore for Redis provides a fully managed in-memory data store service. Cloud Memorystore is compatible with the Redis protocol, allowing easy integration with Spring Caching.

All you have to do is create a Cloud Memorystore instance and use its IP address in `application.properties` file as `spring.redis.host` property value. Everything else is exactly the same as setting up redis-backed Spring caching.

Memorystore instances and your application instances have to be located in the same region.

In short, the following dependencies are needed:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

And then you can use `org.springframework.cache.annotation.Cacheable` annotation for methods you'd like to be cached.

```
@Cacheable("cache1")
public String hello(@PathVariable String name) {
    ....
}
```

If you are interested in a detailed how-to guide, please check Spring Boot Caching using Cloud Memorystore codelab.

Cloud Memorystore documentation can be found here.

Chapter 104. BigQuery

Google Cloud BigQuery is a fully managed, petabyte scale, low cost analytics data warehouse.

Spring Cloud GCP provides:

- A convenience starter which provides autoconfiguration for the `BigQuery` client objects with credentials needed to interface with BigQuery.
- A Spring Integration message handler for loading data into BigQuery tables in your Spring integration pipelines.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-bigquery</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
bigquery'
}
```

104.1. Configuration

The following application properties may be configured with Spring Cloud GCP BigQuery libraries.

Name	Description	Required	Default value
`spring.cloud.gcp.bigquery.datasetName`	The BigQuery dataset that the `BigQueryTemplate` and `BigQueryFileMessageHandler` is scoped to.	Yes	
`spring.cloud.gcp.bigquery.enabled`	Enables or disables Spring Cloud GCP BigQuery autoconfiguration.	No	`true`

spring.cloud.gcp.bigquery.project-id	GCP project ID of the project using BigQuery APIs, if different from the one in the Spring Cloud GCP Core Module.	No	Project ID is typically inferred from gcloud configuration.
spring.cloud.gcp.bigquery.credentials.location	Credentials file location for authenticating with the Google Cloud BigQuery APIs, if different from the ones in the Spring Cloud GCP Core Module	No	Inferred from Application Default Credentials, typically set by gcloud.

104.1.1. BigQuery Client Object

The GcpBigQueryAutoConfiguration class configures an instance of BigQuery for you by inferring your credentials and Project ID from the machine's environment.

Example usage:

```
// BigQuery client object provided by our autoconfiguration.
@Autowired
BigQuery bigquery;

public void runQuery() throws InterruptedException {
  String query = "SELECT column FROM table;";
  QueryJobConfiguration queryConfig =
      QueryJobConfiguration.newBuilder(query).build();

  // Run the query using the BigQuery object
  for (FieldValueList row : bigquery.query(queryConfig).iterateAll()) {
    for (FieldValue val : row) {
      System.out.println(val);
    }
  }
}
```

This object is used to interface with all BigQuery services. For more information, see the BigQuery Client Library usage examples.

104.1.2. BigQueryTemplate

The BigQueryTemplate class is a wrapper over the BigQuery client object and makes it easier to load data into BigQuery tables. A BigQueryTemplate is scoped to a single dataset. The autoconfigured BigQueryTemplate instance will use the dataset provided through the property spring.cloud.gcp.bigquery.datasetName.

Below is a code snippet of how to load a CSV data `InputStream` to a BigQuery table.

```
// BigQuery client object provided by our autoconfiguration.
@Autowired
BigQueryTemplate bigQueryTemplate;

public void loadData(InputStream dataInputStream, String tableName) {
  ListenableFuture<Job> bigQueryJobFuture =
      bigQueryTemplate.writeDataToTable(
          tableName,
          dataFile.getInputStream(),
          FormatOptions.csv());

  // After the future is complete, the data is successfully loaded.
  Job job = bigQueryJobFuture.get();
}
```

104.2. Spring Integration

Spring Cloud GCP BigQuery also provides a Spring Integration message handler `BigQueryFileMessageHandler`. This is useful for incorporating BigQuery data loading operations in a Spring Integration pipeline.

Below is an example configuring a `ServiceActivator` bean using the `BigQueryFileMessageHandler`.

```
@Bean
public DirectChannel bigQueryWriteDataChannel() {
  return new DirectChannel();
}

@Bean
public DirectChannel bigQueryJobReplyChannel() {
  return new DirectChannel();
}

@Bean
@ServiceActivator(inputChannel = "bigQueryWriteDataChannel")
public MessageHandler messageSender(BigQueryTemplate bigQueryTemplate) {
  BigQueryFileMessageHandler messageHandler = new
BigQueryFileMessageHandler(bigQueryTemplate);
  messageHandler.setFormatOptions(FormatOptions.csv());
  messageHandler.setOutputChannel(bigQueryJobReplyChannel());
  return messageHandler;
}
```

104.2.1. BigQuery Message Handling

The `BigQueryFileMessageHandler` accepts the following message payload types for loading into BigQuery: `java.io.File`, `byte[]`, `org.springframework.core.io.Resource`, and `java.io.InputStream`. The message payload will be streamed and written to the BigQuery table you specify.

By default, the `BigQueryFileMessageHandler` is configured to read the headers of the messages it receives to determine how to load the data. The headers are specified by the class `BigQuerySpringMessageHeaders` and summarized below.

Header	Description
`BigQuerySpringMessageHeaders.TABLE_NAME`	Specifies the BigQuery table within your dataset to write to.
`BigQuerySpringMessageHeaders.FORMAT_OPTIONS`	Describes the data format of your data to load (i.e. CSV, JSON, etc.).

Alternatively, you may omit these headers and explicitly set the table name or format options by calling `setTableName(…)` and `setFormatOptions(…)`.

104.2.2. BigQuery Message Reply

After the `BigQueryFileMessageHandler` processes a message to load data to your BigQuery table, it will respond with a `Job` on the reply channel. The Job object provides metadata and information about the load file operation.

By default, the `BigQueryFileMessageHandler` is run in asynchronous mode, with `setSync(false)`, and it will reply with a `ListenableFuture<Job>` on the reply channel. The future is tied to the status of the data loading job and will complete when the job completes.

If the handler is run in synchronous mode with `setSync(true)`, then the handler will block on the completion of the loading job and block until it is complete.

If you decide to use Spring Integration Gateways and you wish to receive `ListenableFuture<Job>` as a reply object in the Gateway, you will have to call `.setAsyncExecutor(null)` on your `GatewayProxyFactoryBean`. This is needed to indicate that you wish to reply on the built-in async support rather than rely on async handling of the gateway.

104.3. Sample

A BigQuery sample application is available.

Chapter 105. Cloud IAP

Cloud Identity-Aware Proxy (IAP) provides a security layer over applications deployed to Google Cloud.

The IAP starter uses Spring Security OAuth 2.0 Resource Server functionality to automatically extract user identity from the proxy-injected `x-goog-iap-jwt-assertion` HTTP header.

The following claims are validated automatically:

- Issue time
- Expiration time
- Issuer
- Audience

The *audience* (`"aud"` claim) validation string is automatically determined when the application is running on App Engine Standard or App Engine Flexible. This functionality relies on Cloud Resource Manager API to retrieve project details, so the following setup is needed:

- Enable Cloud Resource Manager API in GCP Console.
- Make sure your application has `resourcemanager.projects.get` permission.

App Engine automatic *audience* determination can be overridden by using `spring.cloud.gcp.security.iap.audience` property. It supports multiple allowable audiences by providing a comma-delimited list.

For Compute Engine or Kubernetes Engine `spring.cloud.gcp.security.iap.audience` property **must** be provided, as the *audience* string depends on the specific Backend Services setup and cannot be inferred automatically. To determine the *audience* value, follow directions in IAP Verify the JWT payload guide. If `spring.cloud.gcp.security.iap.audience` is not provided, the application will fail to start the following message:

```
No qualifying bean of type
'org.springframework.cloud.gcp.security.iap.AudienceProvider' available.
```

If you create a custom `WebSecurityConfigurerAdapter`, enable extracting user identity by adding `.oauth2ResourceServer().jwt()` configuration to the `HttpSecurity` object. If no custom `WebSecurityConfigurerAdapter` is present, nothing needs to be done because Spring Boot will add this customization by default.

Starter Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-security-iap</artifactId>
</dependency>
```

Starter Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
security-iap'
}
```

105.1. Configuration

The following properties are available.

Modifying registry, algorithm, and header properties might be useful for testing, but the defaults should not be changed in production.

Name	Description	Required	Default
`spring.cloud.gcp.security.iap.registry`	Link to JWK public key registry.	true	www.gstatic.com/iap/verify/public_key-jwk
`spring.cloud.gcp.security.iap.algorithm`	Encryption algorithm used to sign the JWK token.	true	`ES256`
`spring.cloud.gcp.security.iap.header`	Header from which to extract the JWK key.	true	`x-goog-iap-jwt-assertion`
`spring.cloud.gcp.security.iap.issuer`	JWK issuer to verify.	true	cloud.google.com/iap
`spring.cloud.gcp.security.iap.audience`	Custom JWK audience to verify.	false on App Engine; true on GCE/GKE	

105.2. Sample

A sample application is available.

Chapter 106. Cloud Vision

The Google Cloud Vision API allows users to leverage machine learning algorithms for processing images and documents including: image classification, face detection, text extraction, optical character recognition, and others.

Spring Cloud GCP provides:

- A convenience starter which automatically configures authentication settings and client objects needed to begin using the Google Cloud Vision API.
- `CloudVisionTemplate` which simplifies interactions with the Cloud Vision API.
    - Allows you to easily send images to the API as Spring Resources.
    - Offers convenience methods for common operations, such as classifying content of an image.
- `DocumentOcrTemplate` which offers convenient methods for running optical character recognition (OCR) on PDF and TIFF documents.

106.1. Dependency Setup

To begin using this library, add the `spring-cloud-gcp-starter-vision` artifact to your project.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-vision</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-vision'
}
```

106.2. Configuration

The following options may be configured with Spring Cloud GCP Vision libraries.

Name	Description	Required	Default value
`spring.cloud.gcp.vision.enabled`	Enables or disables Cloud Vision autoconfiguration	No	`true`

spring.cloud.gcp.vision.executors-threads-count	Number of threads used during document OCR processing for waiting on long-running OCR operations	No	1
spring.cloud.gcp.vision.json-output-batch-size	Number of document pages to include in each OCR output file.	No	20

106.2.1. Cloud Vision OCR Dependencies

If you are interested in applying optical character recognition (OCR) on documents for your project, you'll need to add both `spring-cloud-gcp-starter-vision` and `spring-cloud-gcp-starter-storage` to your dependencies. The storage starter is necessary because the Cloud Vision API will process your documents and write OCR output files all within your Google Cloud Storage buckets.

Maven coordinates using Spring Cloud GCP BOM:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-vision</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-storage</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-vision'
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-storage'
}
```

106.3. Image Analysis

The `CloudVisionTemplate` allows you to easily analyze images; it provides the following method for interfacing with Cloud Vision:

`public AnnotateImageResponse analyzeImage(Resource imageResource, Feature.Type… featureTypes)`

Parameters:

- `Resource imageResource` refers to the Spring Resource of the image object you wish to analyze. The Google Cloud Vision documentation provides a list of the image types that they support.

- `Feature.Type… featureTypes` refers to a var-arg array of Cloud Vision Features to extract from the image. A feature refers to a kind of image analysis one wishes to perform on an image, such as label detection, OCR recognition, facial detection, etc. One may specify multiple features to analyze within one request. A full list of Cloud Vision Features is provided in the Cloud Vision Feature docs.

Returns:

- `AnnotateImageResponse` contains the results of all the feature analyses that were specified in the request. For each feature type that you provide in the request, `AnnotateImageResponse` provides a getter method to get the result of that feature analysis. For example, if you analyzed an image using the `LABEL_DETECTION` feature, you would retrieve the results from the response using `annotateImageResponse.getLabelAnnotationsList()`.

  `AnnotateImageResponse` is provided by the Google Cloud Vision libraries; please consult the RPC reference or Javadoc for more details. Additionally, you may consult the Cloud Vision docs to familiarize yourself with the concepts and features of the API.

106.3.1. Detect Image Labels Example

Image labeling refers to producing labels that describe the contents of an image. Below is a code sample of how this is done using the Cloud Vision Spring Template.

```
@Autowired
private ResourceLoader resourceLoader;

@Autowired
private CloudVisionTemplate cloudVisionTemplate;

public void processImage() {
  Resource imageResource = this.resourceLoader.getResource("my_image.jpg");
  AnnotateImageResponse response = this.cloudVisionTemplate.analyzeImage(
      imageResource, Type.LABEL_DETECTION);
  System.out.println("Image Classification results: " +
response.getLabelAnnotationsList());
}
```

106.4. Document OCR Template

The `DocumentOcrTemplate` allows you to easily run optical character recognition (OCR) on your PDF and TIFF documents stored in your Google Storage bucket.

First, you will need to create a bucket in Google Cloud Storage and upload the documents you wish to process into the bucket.

106.4.1. Running OCR on a Document

When OCR is run on a document, the Cloud Vision APIs will output a collection of OCR output files

in JSON which describe the text content, bounding rectangles of words and letters, and other information about the document.

The `DocumentOcrTemplate` provides the following method for running OCR on a document saved in Google Cloud Storage:

`ListenableFuture<DocumentOcrResultSet> runOcrForDocument(GoogleStorageLocation document, GoogleStorageLocation outputFilePathPrefix)`

The method allows you to specify the location of the document and the output location for where all the JSON output files will be saved in Google Cloud Storage. It returns a `ListenableFuture` containing `DocumentOcrResultSet` which contains the OCR content of the document.

Running OCR on a document is an operation that can take between several minutes to several hours depending on how large the document is. It is recommended to register callbacks to the returned ListenableFuture or ignore it and process the JSON output files at a later point in time using `readOcrOutputFile` or `readOcrOutputFileSet`.

106.4.2. Running OCR Example

Below is a code snippet of how to run OCR on a document stored in a Google Storage bucket and read the text in the first page of the document.

```
@Autowired
private DocumentOcrTemplate documentOcrTemplate;

public void runOcrOnDocument() {
    GoogleStorageLocation document = GoogleStorageLocation.forFile(
            "your-bucket", "test.pdf");
    GoogleStorageLocation outputLocationPrefix = GoogleStorageLocation.forFolder(
            "your-bucket", "output_folder/test.pdf/");

    ListenableFuture<DocumentOcrResultSet> result =
        this.documentOcrTemplate.runOcrForDocument(
            document, outputLocationPrefix);

    DocumentOcrResultSet ocrPages = result.get(5, TimeUnit.MINUTES);

    String page1Text = ocrPages.getPage(1).getText();
    System.out.println(page1Text);
}
```

106.4.3. Reading OCR Output Files

In some use-cases, you may need to directly read OCR output files stored in Google Cloud Storage.

`DocumentOcrTemplate` offers the following methods for reading and processing OCR output files:

- `readOcrOutputFileSet(GoogleStorageLocation jsonOutputFilePathPrefix)`: Reads a collection of OCR output files under a file path prefix and returns the parsed contents. All of the files under the path should correspond to the same document.
- `readOcrOutputFile(GoogleStorageLocation jsonFile)`: Reads a single OCR output file and returns the parsed contents.

106.4.4. Reading OCR Output Files Example

The code snippet below describes how to read the OCR output files of a single document.

```
@Autowired
private DocumentOcrTemplate documentOcrTemplate;

// Parses the OCR output files corresponding to a single document in a directory
public void parseOutputFileSet() {
  GoogleStorageLocation ocrOutputPrefix = GoogleStorageLocation.forFolder(
      "your-bucket", "json_output_set/");

  DocumentOcrResultSet result =
this.documentOcrTemplate.readOcrOutputFileSet(ocrOutputPrefix);
  System.out.println("Page 2 text: " + result.getPage(2).getText());
}

// Parses a single OCR output file
public void parseSingleOutputFile() {
  GoogleStorageLocation ocrOutputFile = GoogleStorageLocation.forFile(
      "your-bucket", "json_output_set/test_output-2-to-2.json");

  DocumentOcrResultSet result =
this.documentOcrTemplate.readOcrOutputFile(ocrOutputFile);
  System.out.println("Page 2 text: " + result.getPage(2).getText());
}
```

106.5. Sample

Samples are provided to show example usages of Spring Cloud GCP with Google Cloud Vision.

- The Image Labeling Sample shows you how to use image labelling in your Spring application. The application generates labels describing the content inside the images you specify in the application.
- The Document OCR demo shows how you can apply OCR processing on your PDF/TIFF documents in order to extract their text contents.

Chapter 107. Secret Manager

Google Cloud Secret Manager is a secure and convenient method for storing API keys, passwords, certificates, and other sensitive data. A detailed summary of its features can be found in the Secret Manager documentation.

Spring Cloud GCP provides:

- A property source which allows you to specify and load the secrets of your GCP project into your application context as a Bootstrap Property Source.
- A `SecretManagerTemplate` which allows you to read, write, and update secrets in Secret Manager.

107.1. Dependency Setup

To begin using this library, add the `spring-cloud-gcp-starter-secretmanager` artifact to your project.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-secretmanager</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
secretmanager'
}
```

107.1.1. Configuration

By default, Spring Cloud GCP Secret Manager will authenticate using Application Default Credentials. This can be overridden using the authentication properties.

All of the below settings must be specified in a `bootstrap.properties` (or `bootstrap.yaml`) file which is the properties file used to configure settings for bootstrap-phase Spring configuration.

Name	Description	Required	Default value
`spring.cloud.gcp.secretmanager.enabled`	Enables the Secret Manager bootstrap property and template configuration.	No	`true`

spring.cloud.gcp.secretmanager.credentials.location	OAuth2 credentials for authenticating to the Google Cloud Secret Manager API.	No	By default, infers credentials from Application Default Credentials.
spring.cloud.gcp.secretmanager.credentials.encoded-key	Base64-encoded contents of OAuth2 account private key for authenticating to the Google Cloud Secret Manager API.	No	By default, infers credentials from Application Default Credentials.
spring.cloud.gcp.secretmanager.project-id	The default GCP Project used to access Secret Manager API for the template and property source.	No	By default, infers the project from Application Default Credentials.

107.2. Secret Manager Property Source

The Spring Cloud GCP integration for Google Cloud Secret Manager enables you to use Secret Manager as a bootstrap property source.

This allows you to specify and load secrets from Google Cloud Secret Manager as properties into the application context during the Bootstrap Phase, which refers to the initial phase when a Spring application is being loaded.

The Secret Manager property source uses the following syntax to specify secrets:

```
# 1. Long form - specify the project ID, secret ID, and version
sm://projects/<project-id>/secrets/<secret-id>/versions/<version-id>}

# 2.  Long form - specify project ID, secret ID, and use latest version
sm://projects/<project-id>/secrets/<secret-id>

# 3. Short form - specify project ID, secret ID, and version
sm://<project-id>/<secret-id>/<version-id>

# 4. Short form - default project; specify secret + version
#
# The project is inferred from the spring.cloud.gcp.secretmanager.project-id setting
# in your bootstrap.properties (see Configuration) or from application-default
credentials if
# this is not set.
sm://<secret-id>/<version>

# 5. Shortest form - specify secret ID, use default project and latest version.
sm://<secret-id>
```

You can use this syntax in the following places:

1. In your `application.properties` or `bootstrap.properties` files:

```
# Example of the project-secret long-form syntax.
spring.datasource.password=${sm://projects/my-gcp-project/secrets/my-secret}
```

2. Access the value using the `@Value` annotation.

```
// Example of using shortest form syntax.
@Value("${sm://my-secret}")
```

107.3. Secret Manager Template

The `SecretManagerTemplate` class simplifies operations of creating, updating, and reading secrets.

To begin using this class, you may inject an instance of the class using `@Autowired` after adding the starter dependency to your project.

```
@Autowired
private SecretManagerTemplate secretManagerTemplate;
```

Please consult `SecretManagerOperations` for information on what operations are available for the Secret Manager template.

107.4. Sample

A Secret Manager Sample Application is provided which demonstrates basic property source loading and usage of the template class.

Chapter 108. Cloud Runtime Configuration API

The Google Cloud Runtime Configuration service is in **Beta** status, and is only available in snapshot and milestone versions of the project. It’s also not available in the Spring Cloud GCP BOM, unlike other modules.

Spring Cloud GCP makes it possible to use the Google Runtime Configuration API as a Spring Cloud Config server to remotely store your application configuration data.

The Spring Cloud GCP Config support is provided via its own Spring Boot starter. It enables the use of the Google Runtime Configuration API as a source for Spring Boot configuration properties.

Maven coordinates:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-config</artifactId>
    <version>1.2.5.RELEASE</version>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud',
    name: 'spring-cloud-gcp-starter-config',
    version: '1.2.5.RELEASE'
}
```

108.1. Configuration

The following parameters are configurable in Spring Cloud GCP Config:

Name	Description	Required	Default value
`spring.cloud.gcp.config.enabled`	Enables the Config client	No	`false`
`spring.cloud.gcp.config.name`	Name of your application	No	Value of the `spring.application.name` property. If none, `application`

spring.cloud.gcp.config.profile	Active profile	No	Value of the spring.profiles.active property. If more than a single profile, last one is chosen
spring.cloud.gcp.config.timeout-millis	Timeout in milliseconds for connecting to the Google Runtime Configuration API	No	60000
spring.cloud.gcp.config.project-id	GCP project ID where the Google Runtime Configuration API is hosted	No	
spring.cloud.gcp.config.credentials.location	OAuth2 credentials for authenticating with the Google Runtime Configuration API	No	
spring.cloud.gcp.config.credentials.encoded-key	Base64-encoded OAuth2 credentials for authenticating with the Google Runtime Configuration API	No	
spring.cloud.gcp.config.credentials.scopes	OAuth2 scope for Spring Cloud GCP Config credentials	No	www.googleapis.com/auth/cloudruntimeconfig

These properties should be specified in a `bootstrap.yml`/`bootstrap.properties` file, rather than the usual `applications.yml`/`application.properties`.

Core properties, as described in Spring Cloud GCP Core Module, do not apply to Spring Cloud GCP Config.

108.2. Quick start

1. Create a configuration in the Google Runtime Configuration API that is called `${spring.application.name}_${spring.profiles.active}`. In other words, if `spring.application.name` is `myapp` and `spring.profiles.active` is `prod`, the configuration should be called `myapp_prod`.

   In order to do that, you should have the Google Cloud SDK installed, own a Google Cloud Project and run the following command:

```
gcloud init # if this is your first Google Cloud SDK run.
gcloud beta runtime-config configs create myapp_prod
gcloud beta runtime-config configs variables set myapp.queue-size 25 --config-name
myapp_prod
```

2. Configure your `bootstrap.properties` file with your application's configuration data:

```
spring.application.name=myapp
spring.profiles.active=prod
```

3. Add the `@ConfigurationProperties` annotation to a Spring-managed bean:

```
@Component
@ConfigurationProperties("myapp")
public class SampleConfig {

  private int queueSize;

  public int getQueueSize() {
    return this.queueSize;
  }

  public void setQueueSize(int queueSize) {
    this.queueSize = queueSize;
  }
}
```

When your Spring application starts, the `queueSize` field value will be set to 25 for the above `SampleConfig` bean.

108.3. Refreshing the configuration at runtime

Spring Cloud provides support to have configuration parameters be reloadable with the POST request to `/actuator/refresh` endpoint.

1. Add the Spring Boot Actuator dependency:

Maven coordinates:

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

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-actuator'
}
```

2. Add `@RefreshScope` to your Spring configuration class to have parameters be reloadable at runtime.
3. Add `management.endpoints.web.exposure.include=refresh` to your `application.properties` to allow unrestricted access to `/actuator/refresh`.
4. Update a property with `gcloud`:

```
$ gcloud beta runtime-config configs variables set \
  myapp.queue_size 200 \
  --config-name myapp_prod
```

5. Send a POST request to the refresh endpoint:

```
$ curl -XPOST https://myapp.host.com/actuator/refresh
```

108.4. Sample

A sample application and a codelab are available.

Chapter 109. Cloud Foundry

Spring Cloud GCP provides support for Cloud Foundry's GCP Service Broker. Our Pub/Sub, Cloud Spanner, Storage, Stackdriver Trace and Cloud SQL MySQL and PostgreSQL starters are Cloud Foundry aware and retrieve properties like project ID, credentials, etc., that are used in auto configuration from the Cloud Foundry environment.

In order to take advantage of the Cloud Foundry support make sure the following dependency is added:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-cloudfoundry</artifactId>
</dependency>
```

In cases like Pub/Sub's topic and subscription, or Storage's bucket name, where those parameters are not used in auto configuration, you can fetch them using the VCAP mapping provided by Spring Boot. For example, to retrieve the provisioned Pub/Sub topic, you can use the `vcap.services.mypubsub.credentials.topic_name` property from the application environment.

If the same service is bound to the same application more than once, the auto configuration will not be able to choose among bindings and will not be activated for that service. This includes both MySQL and PostgreSQL bindings to the same app.

In order for the Cloud SQL integration to work in Cloud Foundry, auto-reconfiguration must be disabled. You can do so using the `cf set-env <APP> JBP_CONFIG_SPRING_AUTO_RECONFIGURATION '{enabled: false}'` command. Otherwise, Cloud Foundry will produce a `DataSource` with an invalid JDBC URL (i.e., `jdbc:mysql://null/null`).

109.1. User-Provided Services

User-provided services enable developers to use services that are not available in the marketplace with their apps running on Cloud Foundry. For example, you may want to use a user-provided service that points to a shared Google Service (like Cloud Spanner) used across your organization.

In order for Spring Cloud GCP to detect your user-provided service as a Google Cloud Service, you must add an instance tag indicating the Google Cloud Service it uses. The tag should simply be the Cloud Foundry name for the Google Service.

For example, if you create a user-provided service using Cloud Spanner, you might run:

```
$ cf create-user-provided-service user-spanner-service -t "google-spanner" ...
```

This allows Spring Cloud GCP to retrieve the correct service properties from Cloud Foundry and use them in the auto configuration for your application.

A mapping of Google service names to Cloud Foundry names are provided below:

Google Cloud Service	Cloud Foundry Name (add this as a tag)
Google Cloud Pub/Sub	`google-pubsub`
Google Cloud Storage	`google-storage`
Google Cloud Spanner	`google-spanner`
Datastore	`google-datastore`
Firestore	`google-firestore`
BigQuery	`google-bigquery`
Stackdriver Trace	`google-stackdriver-trace`
Cloud Sql (MySQL)	`google-cloudsql-mysql`
Cloud Sql (PostgreSQL)	`google-cloudsql-postgres`

Chapter 110. Kotlin Support

The latest version of the Spring Framework provides first-class support for Kotlin. For Kotlin users of Spring, the Spring Cloud GCP libraries work out-of-the-box and are fully interoperable with Kotlin applications.

For more information on building a Spring application in Kotlin, please consult the Spring Kotlin documentation.

110.1. Prerequisites

Ensure that your Kotlin application is properly set up. Based on your build system, you will need to include the correct Kotlin build plugin in your project:

- Kotlin Maven Plugin
- Kotlin Gradle Plugin

Depending on your application's needs, you may need to augment your build configuration with compiler plugins:

- Kotlin Spring Plugin: Makes your Spring configuration classes/members non-final for convenience.
- Kotlin JPA Plugin: Enables using JPA in Kotlin applications.

Once your Kotlin project is properly configured, the Spring Cloud GCP libraries will work within your application without any additional setup.

110.2. Sample

A Kotlin sample application is provided to demonstrate a working Maven setup and various Spring Cloud GCP integrations from within Kotlin.

Chapter 111. Configuration properties

To see the list of all GCP related configuration properties please check the Appendix page.

Spring Cloud Kubernetes

This reference guide covers how to use Spring Cloud Kubernetes.

Chapter 112. Why do you need Spring Cloud Kubernetes?

Spring Cloud Kubernetes provides implementations of well known Spring Cloud interfaces allowing developers to build and run Spring Cloud applications on Kubernetes. While this project may be useful to you when building a cloud native application, it is also not a requirement in order to deploy a Spring Boot app on Kubernetes. If you are just getting started in your journey to running your Spring Boot app on Kubernetes you can accomplish a lot with nothing more than a basic Spring Boot app and Kubernetes itself. To learn more, you can get started by reading the Spring Boot reference documentation for deploying to Kubernetes and also working through the workshop material Spring and Kubernetes.

Chapter 113. Starters

Starters are convenient dependency descriptors you can include in your application. Include a starter to get the dependencies and Spring Boot auto-configuration for a feature set.

Starter	Features
<dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-kubernetes</artifactId> </dependency>	Discovery Client implementation that resolves service names to Kubernetes Services.
<dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-kubernetes-config</artifactId> </dependency>	Load application properties from Kubernetes ConfigMaps and Secrets. Reload application properties when a ConfigMap or Secret changes.
<dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-kubernetes-ribbon</artifactId> </dependency>	Ribbon client-side load balancer with server list obtained from Kubernetes Endpoints.
<dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-kubernetes-all</artifactId> </dependency>	All Spring Cloud Kubernetes features.

Chapter 114. DiscoveryClient for Kubernetes

This project provides an implementation of Discovery Client for Kubernetes. This client lets you query Kubernetes endpoints (see services) by name. A service is typically exposed by the Kubernetes API server as a collection of endpoints that represent `http` and `https` addresses and that a client can access from a Spring Boot application running as a pod. This discovery feature is also used by the Spring Cloud Kubernetes Ribbon project to fetch the list of the endpoints defined for an application to be load balanced.

This is something that you get for free by adding the following dependency inside your project:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-kubernetes</artifactId>
</dependency>
```

To enable loading of the `DiscoveryClient`, add `@EnableDiscoveryClient` to the according configuration or application class, as the following example shows:

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

Then you can inject the client in your code simply by autowiring it, as the following example shows:

```
@Autowired
private DiscoveryClient discoveryClient;
```

You can choose to enable `DiscoveryClient` from all namespaces by setting the following property in `application.properties`:

```
spring.cloud.kubernetes.discovery.all-namespaces=true
```

If, for any reason, you need to disable the `DiscoveryClient`, you can set the following property in

`application.properties`:

```
spring.cloud.kubernetes.discovery.enabled=false
```

Some Spring Cloud components use the `DiscoveryClient` in order to obtain information about the local service instance. For this to work, you need to align the Kubernetes service name with the `spring.application.name` property.

`spring.application.name` has no effect as far as the name registered for the application within Kubernetes

Spring Cloud Kubernetes can also watch the Kubernetes service catalog for changes and update the `DiscoveryClient` implementation accordingly. In order to enable this functionality you need to add `@EnableScheduling` on a configuration class in your application.

Chapter 115. Kubernetes native service discovery

Kubernetes itself is capable of (server side) service discovery (see: kubernetes.io/docs/concepts/services-networking/service/#discovering-services). Using native kubernetes service discovery ensures compatibility with additional tooling, such as Istio (istio.io), a service mesh that is capable of load balancing, ribbon, circuit breaker, failover, and much more.

The caller service then need only refer to names resolvable in a particular Kubernetes cluster. A simple implementation might use a spring `RestTemplate` that refers to a fully qualified domain name (FQDN), such as `{service-name}.{namespace}.svc.{cluster}.local:{service-port}`.

Additionally, you can use Hystrix for:

- Circuit breaker implementation on the caller side, by annotating the spring boot application class with `@EnableCircuitBreaker`
- Fallback functionality, by annotating the respective method with `@HystrixCommand(fallbackMethod=`

Chapter 116. Kubernetes PropertySource implementations

The most common approach to configuring your Spring Boot application is to create an `application.properties` or `application.yaml` or an `application-profile.properties` or `application-profile.yaml` file that contains key-value pairs that provide customization values to your application or Spring Boot starters. You can override these properties by specifying system properties or environment variables.

116.1. Using a `ConfigMap` `PropertySource`

Kubernetes provides a resource named `ConfigMap` to externalize the parameters to pass to your application in the form of key-value pairs or embedded `application.properties` or `application.yaml` files. The Spring Cloud Kubernetes Config project makes Kubernetes `ConfigMap` instances available during application bootstrapping and triggers hot reloading of beans or Spring context when changes are detected on observed `ConfigMap` instances.

The default behavior is to create a `ConfigMapPropertySource` based on a Kubernetes `ConfigMap` that has a `metadata.name` value of either the name of your Spring application (as defined by its `spring.application.name` property) or a custom name defined within the `bootstrap.properties` file under the following key: `spring.cloud.kubernetes.config.name`.

However, more advanced configuration is possible where you can use multiple `ConfigMap` instances. The `spring.cloud.kubernetes.config.sources` list makes this possible. For example, you could define the following `ConfigMap` instances:

```
spring:
  application:
    name: cloud-k8s-app
  cloud:
    kubernetes:
      config:
        name: default-name
        namespace: default-namespace
        sources:
         # Spring Cloud Kubernetes looks up a ConfigMap named c1 in namespace
default-namespace
         - name: c1
         # Spring Cloud Kubernetes looks up a ConfigMap named default-name in
whatever namespace n2
         - namespace: n2
         # Spring Cloud Kubernetes looks up a ConfigMap named c3 in namespace n3
         - namespace: n3
           name: c3
```

In the preceding example, if `spring.cloud.kubernetes.config.namespace` had not been set, the `ConfigMap` named `c1` would be looked up in the namespace that the application runs.

Any matching `ConfigMap` that is found is processed as follows:

- Apply individual configuration properties.
- Apply as `yaml` the content of any property named `application.yaml`.
- Apply as a properties file the content of any property named `application.properties`.

The single exception to the aforementioned flow is when the `ConfigMap` contains a **single** key that indicates the file is a YAML or properties file. In that case, the name of the key does NOT have to be `application.yaml` or `application.properties` (it can be anything) and the value of the property is treated correctly. This features facilitates the use case where the `ConfigMap` was created by using something like the following:

```
kubectl create configmap game-config --from-file=/path/to/app-config.yaml
```

Assume that we have a Spring Boot application named `demo` that uses the following properties to read its thread pool configuration.

- `pool.size.core`
- `pool.size.maximum`

This can be externalized to config map in `yaml` format as follows:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  pool.size.core: 1
  pool.size.max: 16
```

Individual properties work fine for most cases. However, sometimes, embedded `yaml` is more convenient. In this case, we use a single property named `application.yaml` to embed our `yaml`, as follows:

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yaml: |-
    pool:
      size:
        core: 1
        max:16
```

The following example also works:

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  custom-name.yaml: |-
    pool:
      size:
        core: 1
        max:16
```

You can also configure Spring Boot applications differently depending on active profiles that are merged together when the `ConfigMap` is read. You can provide different property values for different profiles by using an `application.properties` or `application.yaml` property, specifying profile-specific values, each in their own document (indicated by the `---` sequence), as follows:

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yml: |-
    greeting:
      message: Say Hello to the World
    farewell:
      message: Say Goodbye
    ---
    spring:
      profiles: development
    greeting:
      message: Say Hello to the Developers
    farewell:
      message: Say Goodbye to the Developers
    ---
    spring:
      profiles: production
    greeting:
      message: Say Hello to the Ops
```

In the preceding case, the configuration loaded into your Spring Application with the `development` profile is as follows:

```yaml
  greeting:
    message: Say Hello to the Developers
  farewell:
    message: Say Goodbye to the Developers
```

However, if the `production` profile is active, the configuration becomes:

```yaml
  greeting:
    message: Say Hello to the Ops
  farewell:
    message: Say Goodbye
```

If both profiles are active, the property that appears last within the `ConfigMap` overwrites any preceding values.

Another option is to create a different config map per profile and spring boot will automatically

fetch it based on active profiles

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yml: |-
    greeting:
      message: Say Hello to the World
    farewell:
      message: Say Goodbye
```

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo-development
data:
  application.yml: |-
    spring:
      profiles: development
    greeting:
      message: Say Hello to the Developers
    farewell:
      message: Say Goodbye to the Developers
```

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo-production
data:
  application.yml: |-
    spring:
      profiles: production
    greeting:
      message: Say Hello to the Ops
    farewell:
      message: Say Goodbye
```

To tell Spring Boot which `profile` should be enabled at bootstrap, you can pass `SPRING_PROFILES_ACTIVE` environment variable. To do so, you can launch your Spring Boot application with an environment variable that you can define it in the PodSpec at the container

specification. Deployment resource file, as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-name
  labels:
    app: deployment-name
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deployment-name
  template:
    metadata:
      labels:
        app: deployment-name
    spec:
        containers:
        - name: container-name
          image: your-image
          env:
          - name: SPRING_PROFILES_ACTIVE
            value: "development"
```

You should check the security configuration section. To access config maps from inside a pod you need to have the correct Kubernetes service accounts, roles and role bindings.

Another option for using `ConfigMap` instances is to mount them into the Pod by running the Spring Cloud Kubernetes application and having Spring Cloud Kubernetes read them from the file system. This behavior is controlled by the `spring.cloud.kubernetes.config.paths` property. You can use it in addition to or instead of the mechanism described earlier. You can specify multiple (exact) file paths in `spring.cloud.kubernetes.config.paths` by using the `,` delimiter.

You have to provide the full exact path to each property file, because directories are not being recursively parsed.

Table 6. Properties:

Name	Type	Default	Description
`spring.cloud.kubernetes.config.enabled`	`Boolean`	`true`	Enable ConfigMaps `PropertySource`
`spring.cloud.kubernetes.config.name`	`String`	`${spring.application.name}`	Sets the name of `ConfigMap` to look up

Name	Type	Default	Description
`spring.cloud.kubernetes.config.namespace`	`String`	Client namespace	Sets the Kubernetes namespace where to lookup
`spring.cloud.kubernetes.config.paths`	`List`	`null`	Sets the paths where `ConfigMap` instances are mounted
`spring.cloud.kubernetes.config.enableApi`	`Boolean`	`true`	Enable or disable consuming `ConfigMap` instances through APIs

116.2. Secrets PropertySource

Kubernetes has the notion of Secrets for storing sensitive data such as passwords, OAuth tokens, and so on. This project provides integration with `Secrets` to make secrets accessible by Spring Boot applications. You can explicitly enable or disable This feature by setting the `spring.cloud.kubernetes.secrets.enabled` property.

When enabled, the `SecretsPropertySource` looks up Kubernetes for `Secrets` from the following sources:

1. Reading recursively from secrets mounts
2. Named after the application (as defined by `spring.application.name`)
3. Matching some labels

Note:

By default, consuming Secrets through the API (points 2 and 3 above) **is not enabled** for security reasons. The permission 'list' on secrets allows clients to inspect secrets values in the specified namespace. Further, we recommend that containers share secrets through mounted volumes.

If you enable consuming Secrets through the API, we recommend that you limit access to Secrets by using an authorization policy, such as RBAC. For more information about risks and best practices when consuming Secrets through the API refer to this doc.

If the secrets are found, their data is made available to the application.

Assume that we have a spring boot application named `demo` that uses properties to read its database configuration. We can create a Kubernetes secret by using the following command:

```
oc create secret generic db-secret --from-literal=username=user --from
-literal=password=p455w0rd
```

The preceding command would create the following secret (which you can see by using `oc get`

`secrets db-secret -o yaml`):

```
apiVersion: v1
data:
  password: cDQ1NXcwcmQ=
  username: dXNlcg==
kind: Secret
metadata:
  creationTimestamp: 2017-07-04T09:15:57Z
  name: db-secret
  namespace: default
  resourceVersion: "357496"
  selfLink: /api/v1/namespaces/default/secrets/db-secret
  uid: 63c89263-6099-11e7-b3da-76d6186905a8
type: Opaque
```

Note that the data contains Base64-encoded versions of the literal provided by the `create` command.

Your application can then use this secret—for example, by exporting the secret's value as environment variables:

```
apiVersion: v1
kind: Deployment
metadata:
  name: ${project.artifactId}
spec:
   template:
     spec:
       containers:
         - env:
            - name: DB_USERNAME
              valueFrom:
                 secretKeyRef:
                   name: db-secret
                   key: username
            - name: DB_PASSWORD
              valueFrom:
                 secretKeyRef:
                   name: db-secret
                   key: password
```

You can select the Secrets to consume in a number of ways:

1. By listing the directories where secrets are mapped:

```
-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets/db
-secret,etc/secrets/postgresql
```

If you have all the secrets mapped to a common root, you can set them like:

```
-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets
```

2. By setting a named secret:

```
-Dspring.cloud.kubernetes.secrets.name=db-secret
```

3. By defining a list of labels:

```
-Dspring.cloud.kubernetes.secrets.labels.broker=activemq
-Dspring.cloud.kubernetes.secrets.labels.db=postgresql
```

As the case with `ConfigMap`, more advanced configuration is also possible where you can use multiple `Secret` instances. The `spring.cloud.kubernetes.secrets.sources` list makes this possible. For example, you could define the following `Secret` instances:

```
spring:
  application:
    name: cloud-k8s-app
  cloud:
    kubernetes:
      secrets:
        name: default-name
        namespace: default-namespace
        sources:
         # Spring Cloud Kubernetes looks up a Secret named s1 in namespace
default-namespace
         - name: s1
         # Spring Cloud Kubernetes looks up a Secret named default-name in
whatever namespace n2
         - namespace: n2
         # Spring Cloud Kubernetes looks up a Secret named s3 in namespace n3
         - namespace: n3
           name: s3
```

In the preceding example, if `spring.cloud.kubernetes.secrets.namespace` had not been set, the `Secret` named `s1` would be looked up in the namespace that the application runs.

Table 7. Properties:

Name	Type	Default	Description
`spring.cloud.kubernetes.secrets.enabled`	`Boolean`	`true`	Enable Secrets `PropertySource`
`spring.cloud.kubernetes.secrets.name`	`String`	`${spring.application.name}`	Sets the name of the secret to look up
`spring.cloud.kubernetes.secrets.namespace`	`String`	Client namespace	Sets the Kubernetes namespace where to look up
`spring.cloud.kubernetes.secrets.labels`	`Map`	`null`	Sets the labels used to lookup secrets
`spring.cloud.kubernetes.secrets.paths`	`List`	`null`	Sets the paths where secrets are mounted (example 1)
`spring.cloud.kubernetes.secrets.enableApi`	`Boolean`	`false`	Enables or disables consuming secrets through APIs (examples 2 and 3)

Notes:

- The `spring.cloud.kubernetes.secrets.labels` property behaves as defined by Map-based

binding.

- The `spring.cloud.kubernetes.secrets.paths` property behaves as defined by Collection-based binding.
- Access to secrets through the API may be restricted for security reasons. The preferred way is to mount secrets to the Pod.

You can find an example of an application that uses secrets (though it has not been updated to use the new `spring-cloud-kubernetes` project) at spring-boot-camel-config

116.3. `PropertySource` Reload

Some applications may need to detect changes on external property sources and update their internal status to reflect the new configuration. The reload feature of Spring Cloud Kubernetes is able to trigger an application reload when a related `ConfigMap` or `Secret` changes.

By default, this feature is disabled. You can enable it by using the `spring.cloud.kubernetes.reload.enabled=true` configuration property (for example, in the `application.properties` file).

The following levels of reload are supported (by setting the `spring.cloud.kubernetes.reload.strategy` property):

- `refresh` (default): Only configuration beans annotated with `@ConfigurationProperties` or `@RefreshScope` are reloaded. This reload level leverages the refresh feature of Spring Cloud Context.
- `restart_context`: the whole Spring `ApplicationContext` is gracefully restarted. Beans are recreated with the new configuration. In order for the restart context functionality to work properly you must enable and expose the restart actuator endpoint

```
management:
  endpoint:
    restart:
      enabled: true
  endpoints:
    web:
      exposure:
        include: restart
```

- `shutdown`: the Spring `ApplicationContext` is shut down to activate a restart of the container. When you use this level, make sure that the lifecycle of all non-daemon threads is bound to the `ApplicationContext` and that a replication controller or replica set is configured to restart the pod.

Assuming that the reload feature is enabled with default settings (`refresh` mode), the following bean is refreshed when the config map changes:

```
@Configuration
@ConfigurationProperties(prefix = "bean")
public class MyConfig {

    private String message = "a message that can be changed live";

    // getter and setters

}
```

To see that changes effectively happen, you can create another bean that prints the message periodically, as follows

```
@Component
public class MyBean {

    @Autowired
    private MyConfig config;

    @Scheduled(fixedDelay = 5000)
    public void hello() {
        System.out.println("The message is: " + config.getMessage());
    }
}
```

You can change the message printed by the application by using a `ConfigMap`, as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: reload-example
data:
  application.properties: |-
    bean.message=Hello World!
```

Any change to the property named `bean.message` in the `ConfigMap` associated with the pod is reflected in the output. More generally speaking, changes associated to properties prefixed with the value defined by the `prefix` field of the `@ConfigurationProperties` annotation are detected and reflected in the application. Associating a `ConfigMap` with a pod is explained earlier in this chapter.

The full example is available in `spring-cloud-kubernetes-reload-example`.

The reload feature supports two operating modes: * Event (default): Watches for changes in config maps or secrets by using the Kubernetes API (web socket). Any event produces a re-check on the configuration and, in case of changes, a reload. The `view` role on the service account is required in order to listen for config map changes. A higher level role (such as `edit`) is required for secrets (by default, secrets are not monitored). * Polling: Periodically re-creates the configuration from config maps and secrets to see if it has changed. You can configure the polling period by using the `spring.cloud.kubernetes.reload.period` property and defaults to 15 seconds. It requires the same role as the monitored property source. This means, for example, that using polling on file-mounted secret sources does not require particular privileges.

Table 8. Properties:

Name	Type	Default	Description
`spring.cloud.kubernetes.reload.enabled`	`Boolean`	`false`	Enables monitoring of property sources and configuration reload
`spring.cloud.kubernetes.reload.monitoring-config-maps`	`Boolean`	`true`	Allow monitoring changes in config maps
`spring.cloud.kubernetes.reload.monitoring-secrets`	`Boolean`	`false`	Allow monitoring changes in secrets
`spring.cloud.kubernetes.reload.strategy`	`Enum`	`refresh`	The strategy to use when firing a reload (`refresh`, `restart_context`, or `shutdown`)
`spring.cloud.kubernetes.reload.mode`	`Enum`	`event`	Specifies how to listen for changes in property sources (`event` or `polling`)
`spring.cloud.kubernetes.reload.period`	`Duration`	`15s`	The period for verifying changes when using the `polling` strategy

Notes: * You should not use properties under `spring.cloud.kubernetes.reload` in config maps or secrets. Changing such properties at runtime may lead to unexpected results. * Deleting a property or the whole config map does not restore the original state of the beans when you use the `refresh` level.

Chapter 117. Ribbon Discovery in Kubernetes

Spring Cloud client applications that call a microservice should be interested on relying on a client load-balancing feature in order to automatically discover at which endpoint(s) it can reach a given service. This mechanism has been implemented within the spring-cloud-kubernetes-ribbon project, where a Kubernetes client populates a Ribbon `ServerList` that contains information about such endpoints.

The implementation is part of the following starter that you can use by adding its dependency to your pom file:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-kubernetes-ribbon</artifactId>
    <version>${latest.version}</version>
</dependency>
```

When the list of the endpoints is populated, the Kubernetes client searches the registered endpoints that live in the current namespace or project by matching the service name defined in the Ribbon Client annotation, as follows:

```
@RibbonClient(name = "name-service")
```

You can configure Ribbon's behavior by providing properties in your `application.properties` (through your application's dedicated `ConfigMap`) by using the following format: `<name of your service>.ribbon.<Ribbon configuration key>`, where:

- `<name of your service>` corresponds to the service name you access over Ribbon, as configured by using the `@RibbonClient` annotation (such as `name-service` in the preceding example).
- `<Ribbon configuration key>` is one of the Ribbon configuration keys defined by Ribbon's `CommonClientConfigKey` class.

Additionally, the `spring-cloud-kubernetes-ribbon` project defines two additional configuration keys to further control how Ribbon interacts with Kubernetes. In particular, if an endpoint defines multiple ports, the default behavior is to use the first one found. To select more specifically which port to use in a multi-port service, you can use the `PortName` key. If you want to specify in which Kubernetes namespace the target service should be looked up, you can use the `KubernetesNamespace` key, remembering in both instances to prefix these keys with your service name and `ribbon` prefix, as specified earlier.

Table 9. Spring Cloud Kubernetes Ribbon Configuration

Property Key	Type	Default Value
spring.cloud.kubernetes.ribbon.enabled	boolean	true
spring.cloud.kubernetes.ribbon.mode	`KubernetesRibbonMode`	POD
spring.cloud.kubernetes.ribbon.cluster-domain	string	cluster.local

- `spring.cloud.kubernetes.ribbon.mode` supports `POD` and `SERVICE` modes.
  - The POD mode is to achieve load balancing by obtaining the Pod IP address of Kubernetes and using Ribbon. POD mode uses the load balancing of the Ribbon Does not support Kubernetes load balancing, The traffic policy of `Istio` is not supported.
  - the `SERVICE` mode is directly based on the `service name` of the Ribbon. Get The Kubernetes service is concatenated into `service-name.{namespace}.svc.{cluster.domain}:{port}` such as: `demo1.default.svc.cluster.local:8080`. the `SERVICE` mode uses load balancing of the Kubernetes service to support Istio's traffic policy.
- `spring.cloud.kubernetes.ribbon.cluster-domain` Set the custom Kubernetes cluster domain suffix. The default value is: 'cluster.local'

The following examples use this module for ribbon discovery:

- Spring Cloud Circuitbreaker and Ribbon
- fabric8-quickstarts - Spring Boot - Ribbon
- Kubeflix - LoanBroker - Bank

You can disable the Ribbon discovery client by setting the `spring.cloud.kubernetes.ribbon.enabled=false` key within the application properties file.

Chapter 118. Kubernetes Ecosystem Awareness

All of the features described earlier in this guide work equally well, regardless of whether your application is running inside Kubernetes. This is really helpful for development and troubleshooting. From a development point of view, this lets you start your Spring Boot application and debug one of the modules that is part of this project. You need not deploy it in Kubernetes, as the code of the project relies on the Fabric8 Kubernetes Java client, which is a fluent DSL that can communicate by using `http` protocol to the REST API of the Kubernetes Server.

To disable the integration with Kubernetes you can set `spring.cloud.kubernetes.enabled` to `false`. Please be aware that when `spring-cloud-kubernetes-config` is on the classpath, `spring.cloud.kubernetes.enabled` should be set in `bootstrap.{properties|yml}` (or the profile specific one) otherwise it should be in `application.{properties|yml}` (or the profile specific one). Also note that these properties: `spring.cloud.kubernetes.config.enabled` and `spring.cloud.kubernetes.secrets.enabled` only take effect when set in `bootstrap.{properties|yml}`

118.1. Kubernetes Profile Autoconfiguration

When the application runs as a pod inside Kubernetes, a Spring profile named `kubernetes` automatically gets activated. This lets you customize the configuration, to define beans that are applied when the Spring Boot application is deployed within the Kubernetes platform (for example, different development and production configuration).

118.2. Istio Awareness

When you include the `spring-cloud-kubernetes-istio` module in the application classpath, a new profile is added to the application, provided the application is running inside a Kubernetes Cluster with Istio installed. You can then use spring `@Profile("istio")` annotations in your Beans and `@Configuration` classes.

The Istio awareness module uses `me.snowdrop:istio-client` to interact with Istio APIs, letting us discover traffic rules, circuit breakers, and so on, making it easy for our Spring Boot applications to consume this data to dynamically configure themselves according to the environment.

Chapter 119. Pod Health Indicator

Spring Boot uses `HealthIndicator` to expose info about the health of an application. That makes it really useful for exposing health-related information to the user and makes it a good fit for use as readiness probes.

The Kubernetes health indicator (which is part of the core module) exposes the following info:

- Pod name, IP address, namespace, service account, node name, and its IP address
- A flag that indicates whether the Spring Boot application is internal or external to Kubernetes

Chapter 120. Leader Election

<TBD>

Chapter 121. LoadBalancer for Kubernetes

This project includes Spring Cloud Load Balancer for load balancing based on Kubernetes Endpoints and provides implementation of load balancer based on Kubernetes Service. To include it to your project add the following dependency.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-kubernetes-loadbalancer</artifactId>
</dependency>
```

To enable load balancing based on Kubernetes Service name use the following property. Then load balancer would try to call application using address, for example `service-a.default.svc.cluster.local`

```
spring.cloud.kubernetes.loadbalancer.mode=SERVICE
```

To enabled load balancing across all namespaces use the following property. Property from `spring-cloud-kubernetes-discovery` module is respected.

```
spring.cloud.kubernetes.discovery.all-namespaces=true
```

Chapter 122. Security Configurations Inside Kubernetes

122.1. Namespace

Most of the components provided in this project need to know the namespace. For Kubernetes (1.3+), the namespace is made available to the pod as part of the service account secret and is automatically detected by the client. For earlier versions, it needs to be specified as an environment variable to the pod. A quick way to do this is as follows:

```
env:
- name: "KUBERNETES_NAMESPACE"
  valueFrom:
    fieldRef:
      fieldPath: "metadata.namespace"
```

122.2. Service Account

For distributions of Kubernetes that support more fine-grained role-based access within the cluster, you need to make sure a pod that runs with `spring-cloud-kubernetes` has access to the Kubernetes API. For any service accounts you assign to a deployment or pod, you need to make sure they have the correct roles.

Depending on the requirements, you'll need `get`, `list` and `watch` permission on the following resources:

Table 10. Kubernetes Resource Permissions

Dependency	Resources
spring-cloud-starter-kubernetes	pods, services, endpoints
spring-cloud-starter-kubernetes-config	configmaps, secrets
spring-cloud-starter-kubernetes-ribbon	pods, services, endpoints

For development purposes, you can add `cluster-reader` permissions to your `default` service account. On a production system you'll likely want to provide more granular permissions.

The following Role and RoleBinding are an example for namespaced permissions for the `default` account:

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: YOUR-NAME-SPACE
  name: namespace-reader
rules:
  - apiGroups: ["", "extensions", "apps"]
    resources: ["configmaps", "pods", "services", "endpoints", "secrets"]
    verbs: ["get", "list", "watch"]

---

kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: namespace-reader-binding
  namespace: YOUR-NAME-SPACE
subjects:
- kind: ServiceAccount
  name: default
  apiGroup: ""
roleRef:
  kind: Role
  name: namespace-reader
  apiGroup: ""
```

Chapter 123. Service Registry Implementation

In Kubernetes service registration is controlled by the platform, the application itself does not control registration as it may do in other platforms. For this reason using `spring.cloud.service-registry.auto-registration.enabled` or setting `@EnableDiscoveryClient(autoRegister=false)` will have no effect in Spring Cloud Kubernetes.

Chapter 124. Examples

Spring Cloud Kubernetes tries to make it transparent for your applications to consume Kubernetes Native Services by following the Spring Cloud interfaces.

In your applications, you need to add the `spring-cloud-kubernetes-discovery` dependency to your classpath and remove any other dependency that contains a `DiscoveryClient` implementation (that is, a Eureka discovery client). The same applies for `PropertySourceLocator`, where you need to add to the classpath the `spring-cloud-kubernetes-config` and remove any other dependency that contains a `PropertySourceLocator` implementation (that is, a configuration server client).

The following projects highlight the usage of these dependencies and demonstrate how you can use these libraries from any Spring Boot application:

- Spring Cloud Kubernetes Examples: the ones located inside this repository.
- Spring Cloud Kubernetes Full Example: Minions and Boss
  - Minion
  - Boss
- Spring Cloud Kubernetes Full Example: SpringOne Platform Tickets Service
- Spring Cloud Gateway with Spring Cloud Kubernetes Discovery and Config
- Spring Boot Admin with Spring Cloud Kubernetes Discovery and Config

Chapter 125. Other Resources

This section lists other resources, such as presentations (slides) and videos about Spring Cloud Kubernetes.

- S1P Spring Cloud on PKS
- Spring Cloud, Docker, Kubernetes → London Java Community July 2018

Please feel free to submit other resources through pull requests to this repository.

Chapter 126. Configuration properties

To see the list of all Sleuth related configuration properties please check the Appendix page.

Chapter 127. Building

127.1. Basic Compile and Test

To build the source you will need to install JDK 1.7.

Spring Cloud uses Maven for most build-related activities, and you should be able to get off the ground quite quickly by cloning the project you are interested in and typing

```
$ ./mvnw install
```

You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.

Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

For hints on how to build the project look in `.travis.yml` if there is one. There should be a "script" and maybe "install" command. Also look at the "services" section to see if any services need to be running locally (e.g. mongo or rabbit). Ignore the git-related bits that you might find in "before_install" since they're related to setting git credentials and you already have those.

The projects that require middleware generally include a `docker-compose.yml`, so consider using Docker Compose to run the middeware servers in Docker containers. See the README in the scripts demo repository for specific instructions about the common cases of mongo, rabbit and redis.

If all else fails, build with the command from `.travis.yml` (usually `./mvnw install`).

127.2. Documentation

The spring-cloud-build module has a "docs" profile, and if you switch that on it will try to build asciidoc sources from `src/main/asciidoc`. As part of that process it will look for a `README.adoc` and process it by loading all the includes, but not parsing or rendering it, just copying it to `${main.basedir}` (defaults to `$/tmp/releaser-1598648275631-0/spring-cloud-release/train-docs/target/unpacked-docs`, i.e. the root of the project). If there are any changes in the README it will then show up after a Maven build as a modified file in the correct place. Just commit it and push the change.

127.3. Working with the code

If you don’t have an IDE preference we would recommend that you use Spring Tools Suite or Eclipse when working with the code. We use the m2eclipse eclipse plugin for maven support. Other IDEs and tools should also work without issue as long as they use Maven 3.3.3 or better.

127.3.1. Activate the Spring Maven profile

Spring Cloud projects require the 'spring' Maven profile to be activated to resolve the spring milestone and snapshot repositories. Use your preferred IDE to set this profile to be active, or you may experience build errors.

127.3.2. Importing into eclipse with m2eclipse

We recommend the m2eclipse eclipse plugin when working with eclipse. If you don’t already have m2eclipse installed it is available from the "eclipse marketplace".

Older versions of m2e do not support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the right profile for the projects. If you see many different errors related to the POMs in the projects, check that you have an up to date installation. If you can’t upgrade m2e, add the "spring" profile to your `settings.xml`. Alternatively you can copy the repository settings from the "spring" profile of the parent pom into your `settings.xml`.

127.3.3. Importing into eclipse without m2eclipse

If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu.

Chapter 128. Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

128.1. Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the Contributor License Agreement. Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

128.2. Code of Conduct

This project adheres to the Contributor Covenant code of conduct. By participating, you are expected to uphold this code. Please report unacceptable behavior to spring-code-of-conduct@pivotal.io.

128.3. Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the Spring Cloud Build project. If using IntelliJ, you can use the Eclipse Code Formatter Plugin to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the .java files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).
- When writing a commit message please follow these conventions, if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).

128.4. Checkstyle

Spring Cloud Build comes with a set of checkstyle rules. You can find them in the `spring-cloud-build-tools` module. The most notable files under the module are:

spring-cloud-build-tools/

```
└── src
    ├── checkstyle
    │   └── checkstyle-suppressions.xml ③
    └── main
        └── resources
            ├── checkstyle-header.txt ②
            └── checkstyle.xml ①
```

① Default Checkstyle rules

② File header setup

③ Default suppression rules

128.4.1. Checkstyle configuration

Checkstyle rules are **disabled by default**. To add checkstyle to your project just define the following properties and plugins.

pom.xml

```
<properties>
<maven-checkstyle-plugin.failsOnError>true</maven-checkstyle-plugin.failsOnError> ①
        <maven-checkstyle-plugin.failsOnViolation>true
        </maven-checkstyle-plugin.failsOnViolation> ②
        <maven-checkstyle-plugin.includeTestSourceDirectory>true
        </maven-checkstyle-plugin.includeTestSourceDirectory> ③
</properties>

<build>
        <plugins>
            <plugin> ④
                <groupId>io.spring.javaformat</groupId>
                <artifactId>spring-javaformat-maven-plugin</artifactId>
            </plugin>
            <plugin> ⑤
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
            </plugin>
        </plugins>

    <reporting>
        <plugins>
            <plugin> ⑤
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
            </plugin>
        </plugins>
    </reporting>
</build>
```

① Fails the build upon Checkstyle errors

② Fails the build upon Checkstyle violations

③ Checkstyle analyzes also the test sources

④ Add the Spring Java Format plugin that will reformat your code to pass most of the Checkstyle formatting rules

⑤ Add checkstyle plugin to your build and reporting phases

If you need to suppress some rules (e.g. line length needs to be longer), then it's enough for you to define a file under `${project.root}/src/checkstyle/checkstyle-suppressions.xml` with your suppressions. Example:

projectRoot/src/checkstyle/checkstyle-suppresions.xml

```
<?xml version="1.0"?>
<!DOCTYPE suppressions PUBLIC
        "-//Puppy Crawl//DTD Suppressions 1.1//EN"
        "https://www.puppycrawl.com/dtds/suppressions_1_1.dtd">
<suppressions>
    <suppress files=".*ConfigServerApplication\.java"
checks="HideUtilityClassConstructor"/>
    <suppress files=".*ConfigClientWatch\.java" checks="LineLengthCheck"/>
</suppressions>
```

It's advisable to copy the `${spring-cloud-build.rootFolder}/.editorconfig` and `${spring-cloud-build.rootFolder}/.springformat` to your project. That way, some default formatting rules will be applied. You can do so by running this script:

```
$ curl https://raw.githubusercontent.com/spring-cloud/spring-cloud-
build/master/.editorconfig -o .editorconfig
$ touch .springformat
```

128.5. IDE setup

128.5.1. Intellij IDEA

In order to setup Intellij you should import our coding conventions, inspection profiles and set up the checkstyle plugin. The following files can be found in the Spring Cloud Build project.

spring-cloud-build-tools/

```
└── src
    ├── checkstyle
    │   └── checkstyle-suppressions.xml ③
    └── main
        └── resources
            ├── checkstyle-header.txt ②
            ├── checkstyle.xml ①
            └── intellij
                ├── Intellij_Project_Defaults.xml ④
                └── Intellij_Spring_Boot_Java_Conventions.xml ⑤
```

① Default Checkstyle rules

② File header setup

③ Default suppression rules

④ Project defaults for Intellij that apply most of Checkstyle rules

⑤ Project style conventions for Intellij that apply most of Checkstyle rules

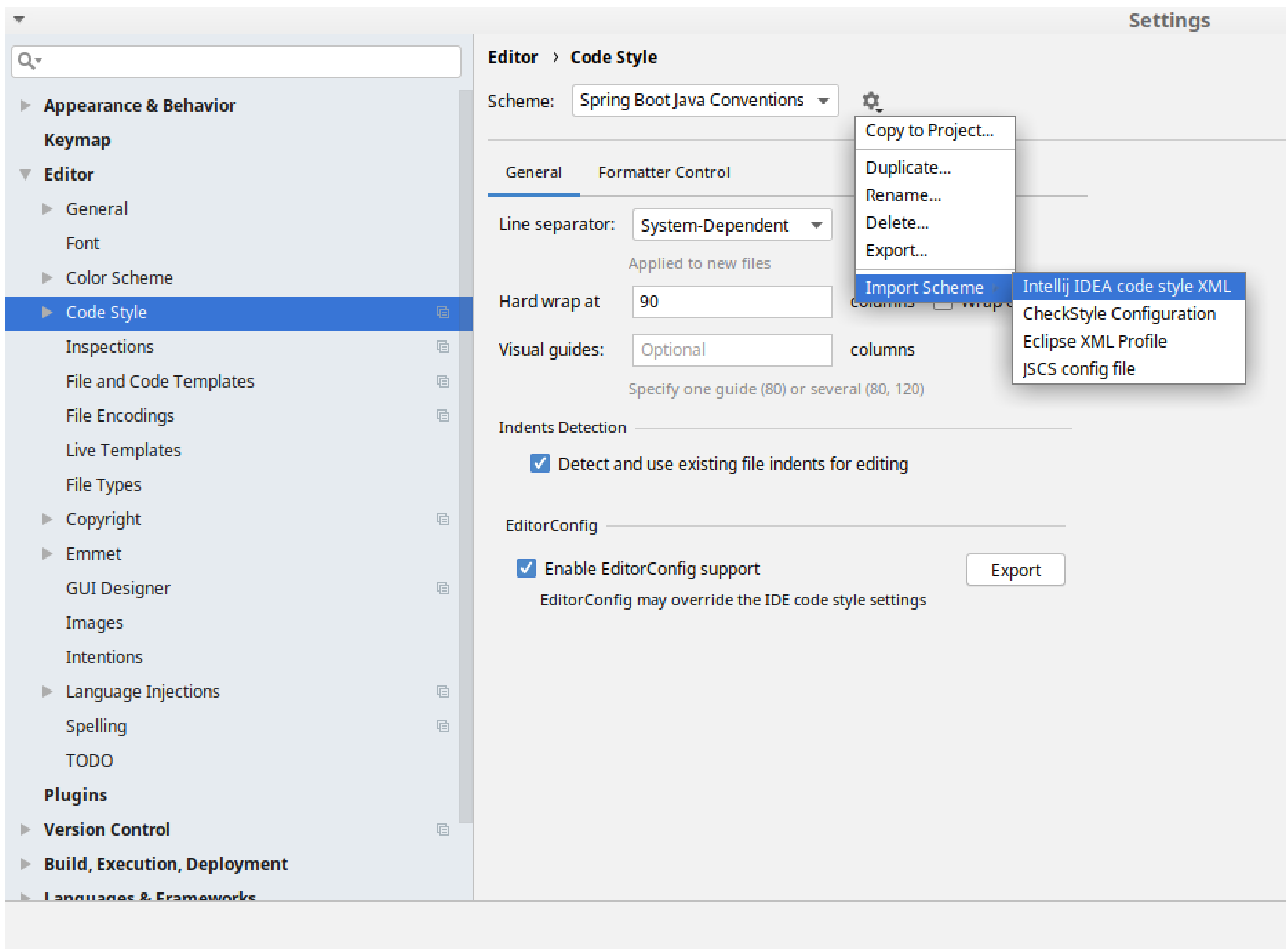


Figure 5. Code style

Go to `File` → `Settings` → `Editor` → `Code style`. There click on the icon next to the `Scheme` section. There, click on the `Import Scheme` value and pick the `Intellij IDEA code style XML` option. Import the `spring-cloud-build-tools/src/main/resources/intellij/Intellij_Spring_Boot_Java_Conventions.xml` file.

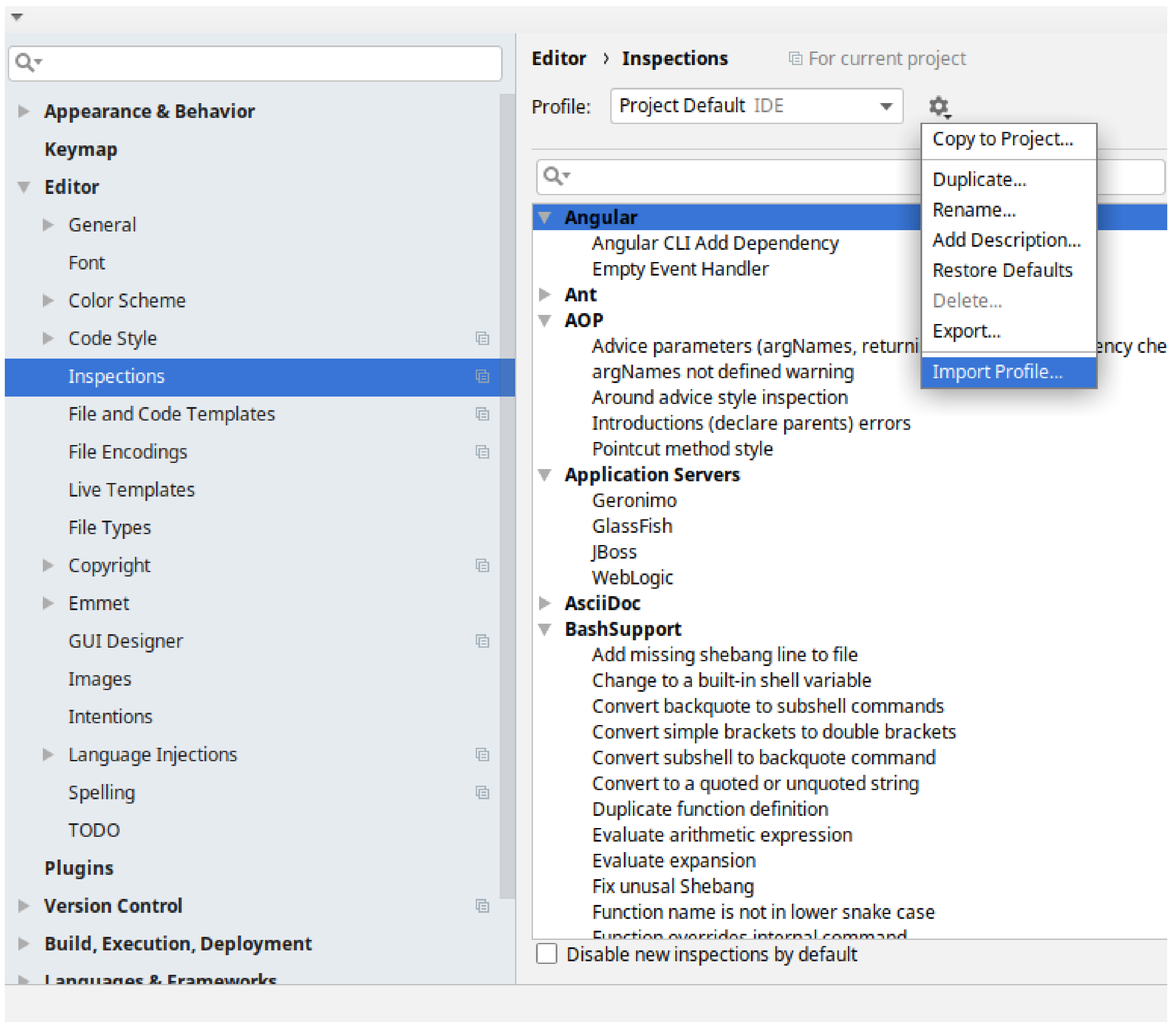
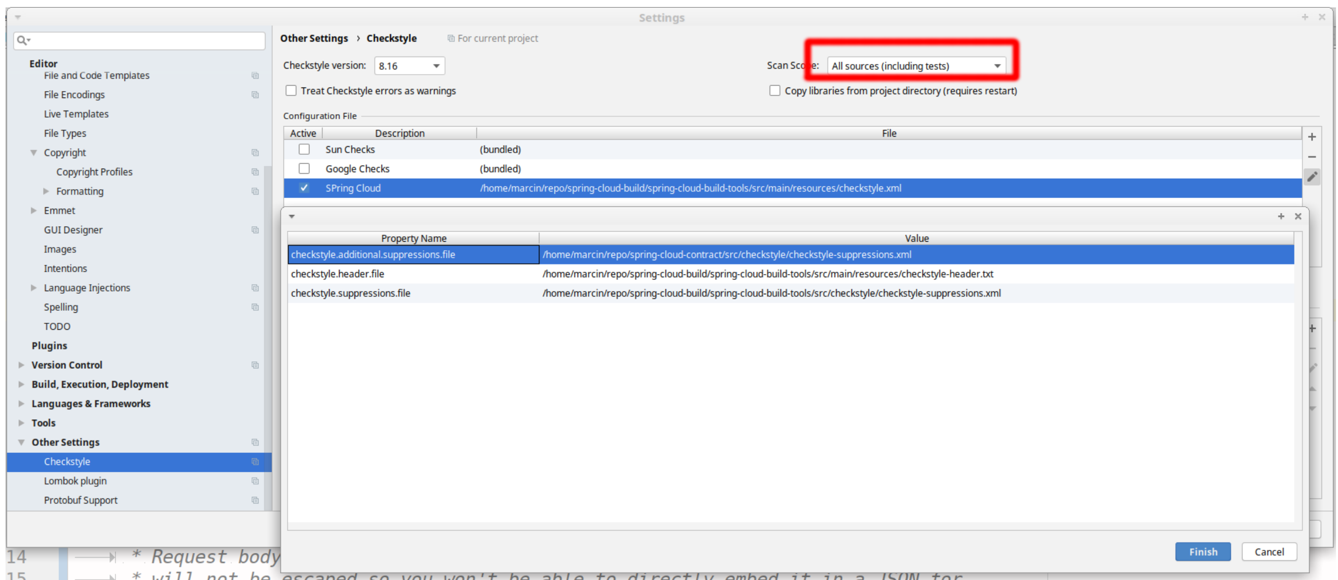


Figure 6. Inspection profiles

Go to `File` → `Settings` → `Editor` → `Inspections`. There click on the icon next to the `Profile` section. There, click on the `Import Profile` and import the `spring-cloud-build-tools/src/main/resources/intellij/Intellij_Project_Defaults.xml` file.

Checkstyle

To have Intellij work with Checkstyle, you have to install the `Checkstyle` plugin. It's advisable to also install the `Assertions2Assertj` to automatically convert the JUnit assertions



Go to `File` → `Settings` → `Other settings` → `Checkstyle`. There click on the `+` icon in the `Configuration file` section. There, you'll have to define where the checkstyle rules should be picked from. In the image above, we've picked the rules from the cloned Spring Cloud Build repository. However, you can point to the Spring Cloud Build's GitHub repository (e.g. for the `checkstyle.xml` : `raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle.xml`). We need to provide the following variables:

- `checkstyle.header.file` - please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/main/resources/checkstyle-header.txt` file either in your cloned repo or via the `raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle-header.txt` URL.
- `checkstyle.suppressions.file` - default suppressions. Please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml` file either in your cloned repo or via the `raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml` URL.
- `checkstyle.additional.suppressions.file` - this variable corresponds to suppressions in your local project. E.g. you're working on `spring-cloud-contract`. Then point to the `project-root/src/checkstyle/checkstyle-suppressions.xml` folder. Example for `spring-cloud-contract` would be: `/home/username/spring-cloud-contract/src/checkstyle/checkstyle-suppressions.xml`.

Remember to set the `Scan Scope` to `All sources` since we apply checkstyle rules for production and test sources.

Spring Cloud Netflix

Hoxton.SR8

This project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon).

Chapter 129. Service Discovery: Eureka Clients

Service Discovery is one of the key tenets of a microservice-based architecture. Trying to hand-configure each client or some form of convention can be difficult to do and can be brittle. Eureka is the Netflix Service Discovery Server and Client. The server can be configured and deployed to be highly available, with each server replicating state about the registered services to the others.

129.1. How to Include Eureka Client

To include the Eureka Client in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-netflix-eureka-client`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

129.2. Registering with Eureka

When a client registers with Eureka, it provides meta-data about itself — such as host, port, health indicator URL, home page, and other details. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally removed from the registry.

The following example shows a minimal Eureka client application:

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

Note that the preceding example shows a normal Spring Boot application. By having `spring-cloud-starter-netflix-eureka-client` on the classpath, your application automatically registers with the Eureka Server. Configuration is required to locate the Eureka server, as shown in the following example:

application.yml

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

In the preceding example, `defaultZone` is a magic string fallback value that provides the service URL for any client that does not express a preference (in other words, it is a useful default).

The `defaultZone` property is case sensitive and requires camel case because the `serviceUrl` property is a `Map<String, String>`. Therefore, the `defaultZone` property does not follow the normal Spring Boot snake-case convention of `default-zone`.

The default application name (that is, the service ID), virtual host, and non-secure port (taken from the `Environment`) are `${spring.application.name}`, `${spring.application.name}` and `${server.port}`, respectively.

Having `spring-cloud-starter-netflix-eureka-client` on the classpath makes the app into both a Eureka “instance” (that is, it registers itself) and a “client” (it can query the registry to locate other services). The instance behaviour is driven by `eureka.instance.*` configuration keys, but the defaults are fine if you ensure that your application has a value for `spring.application.name` (this is the default for the Eureka service ID or VIP).

See EurekaInstanceConfigBean and EurekaClientConfigBean for more details on the configurable options.

To disable the Eureka Discovery Client, you can set `eureka.client.enabled` to `false`. Eureka Discovery Client will also be disabled when `spring.cloud.discovery.enabled` is set to `false`.

129.3. Authenticating with the Eureka Server

HTTP basic authentication is automatically added to your eureka client if one of the `eureka.client.serviceUrl.defaultZone` URLs has credentials embedded in it (curl style, as follows: `user:password@localhost:8761/eureka`). For more complex needs, you can create a `@Bean` of type `DiscoveryClientOptionalArgs` and inject `ClientFilter` instances into it, all of which is applied to the calls from the client to the server.

Because of a limitation in Eureka, it is not possible to support per-server basic auth credentials, so only the first set that are found is used.

When Eureka server requires client side certificate for authentication, the client side certificate and trust store can be configured via properties, as shown in following example:

application.yml

```
eureka:
  client:
    tls:
      enabled: true
      key-store: <path-of-key-store>
      key-store-type: PKCS12
      key-store-password: <key-store-password>
      key-password: <key-password>
      trust-store: <path-of-trust-store>
      trust-store-type: PKCS12
      trust-store-password: <trust-store-password>
```

The `eureka.client.tls.enabled` needs to be true to enable Eureka client side TLS. When `eureka.client.tls.trust-store` is omitted, a JVM default trust store is used. The default value for `eureka.client.tls.key-store-type` and `eureka.client.tls.trust-store-type` is PKCS12. When password properties are omitted, empty password is assumed.

129.4. Status Page and Health Indicator

The status page and health indicators for a Eureka instance default to `/info` and `/health` respectively, which are the default locations of useful endpoints in a Spring Boot Actuator application. You need to change these, even for an Actuator application if you use a non-default context path or servlet path (such as `server.servletPath=/custom`). The following example shows the default values for the two settings:

application.yml

```
eureka:
  instance:
    statusPageUrlPath: ${server.servletPath}/info
    healthCheckUrlPath: ${server.servletPath}/health
```

These links show up in the metadata that is consumed by clients and are used in some scenarios to decide whether to send requests to your application, so it is helpful if they are accurate.

In Dalston it was also required to set the status and health check URLs when changing that management context path. This requirement was removed beginning in Edgware.

129.5. Registering a Secure Application

If your app wants to be contacted over HTTPS, you can set two flags in the `EurekaInstanceConfig`:

- `eureka.instance.[nonSecurePortEnabled]=[false]`
- `eureka.instance.[securePortEnabled]=[true]`

Doing so makes Eureka publish instance information that shows an explicit preference for secure communication. The Spring Cloud `DiscoveryClient` always returns a URI starting with `https` for a service configured this way. Similarly, when a service is configured this way, the Eureka (native) instance information has a secure health check URL.

Because of the way Eureka works internally, it still publishes a non-secure URL for the status and home pages unless you also override those explicitly. You can use placeholders to configure the eureka instance URLs, as shown in the following example:

application.yml

```
eureka:
  instance:
    statusPageUrl: https://${eureka.hostname}/info
    healthCheckUrl: https://${eureka.hostname}/health
    homePageUrl: https://${eureka.hostname}/
```

(Note that `${eureka.hostname}` is a native placeholder only available in later versions of Eureka. You could achieve the same thing with Spring placeholders as well — for example, by using `${eureka.instance.hostName}`.)

If your application runs behind a proxy, and the SSL termination is in the proxy (for example, if you run in Cloud Foundry or other platforms as a service), then you need to ensure that the proxy “forwarded” headers are intercepted and handled by the application. If the Tomcat container embedded in a Spring Boot application has explicit configuration for the 'X-Forwarded-*` headers, this happens automatically. The links rendered by your app to itself being wrong (the wrong host, port, or protocol) is a sign that you got this configuration wrong.

129.6. Eureka’s Health Checks

By default, Eureka uses the client heartbeat to determine if a client is up. Unless specified otherwise, the Discovery Client does not propagate the current health check status of the application, per the Spring Boot Actuator. Consequently, after successful registration, Eureka always announces that the application is in 'UP' state. This behavior can be altered by enabling Eureka health checks, which results in propagating application status to Eureka. As a consequence, every other application does not send traffic to applications in states other then 'UP'. The following example shows how to enable health checks for the client:

application.yml

```
eureka:
  client:
    healthcheck:
      enabled: true
```

`eureka.client.healthcheck.enabled=true` should only be set in `application.yml`. Setting the value in `bootstrap.yml` causes undesirable side effects, such as registering in Eureka with an `UNKNOWN` status.

If you require more control over the health checks, consider implementing your own `com.netflix.appinfo.HealthCheckHandler`.

129.7. Eureka Metadata for Instances and Clients

It is worth spending a bit of time understanding how the Eureka metadata works, so you can use it in a way that makes sense in your platform. There is standard metadata for information such as hostname, IP address, port numbers, the status page, and health check. These are published in the service registry and used by clients to contact the services in a straightforward way. Additional metadata can be added to the instance registration in the `eureka.instance.metadataMap`, and this metadata is accessible in the remote clients. In general, additional metadata does not change the behavior of the client, unless the client is made aware of the meaning of the metadata. There are a couple of special cases, described later in this document, where Spring Cloud already assigns meaning to the metadata map.

129.7.1. Using Eureka on Cloud Foundry

Cloud Foundry has a global router so that all instances of the same app have the same hostname (other PaaS solutions with a similar architecture have the same arrangement). This is not necessarily a barrier to using Eureka. However, if you use the router (recommended or even mandatory, depending on the way your platform was set up), you need to explicitly set the hostname and port numbers (secure or non-secure) so that they use the router. You might also want to use instance metadata so that you can distinguish between the instances on the client (for example, in a custom load balancer). By default, the `eureka.instance.instanceId` is `vcap.application.instance_id`, as shown in the following example:

application.yml

```
eureka:
  instance:
    hostname: ${vcap.application.uris[0]}
    nonSecurePort: 80
```

Depending on the way the security rules are set up in your Cloud Foundry instance, you might be able to register and use the IP address of the host VM for direct service-to-service calls. This feature is not yet available on Pivotal Web Services (PWS).

129.7.2. Using Eureka on AWS

If the application is planned to be deployed to an AWS cloud, the Eureka instance must be configured to be AWS-aware. You can do so by customizing the EurekaInstanceConfigBean as follows:

```
@Bean
@Profile("!default")
public EurekaInstanceConfigBean eurekaInstanceConfig(InetUtils inetUtils) {
  EurekaInstanceConfigBean b = new EurekaInstanceConfigBean(inetUtils);
  AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");
  b.setDataCenterInfo(info);
  return b;
}
```

129.7.3. Changing the Eureka Instance ID

A vanilla Netflix Eureka instance is registered with an ID that is equal to its host name (that is, there is only one service per host). Spring Cloud Eureka provides a sensible default, which is defined as follows:

`${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance_id:${server.port}}}`

An example is `myhost:myappname:8080`.

By using Spring Cloud, you can override this value by providing a unique identifier in `eureka.instance.instanceId`, as shown in the following example:

application.yml

```
eureka:
  instance:
    instanceId:
${spring.application.name}:${vcap.application.instance_id:${spring.application.instanc
e_id:${random.value}}}
```

With the metadata shown in the preceding example and multiple service instances deployed on localhost, the random value is inserted there to make the instance unique. In Cloud Foundry, the `vcap.application.instance_id` is populated automatically in a Spring Boot application, so the random value is not needed.

129.8. Using the EurekaClient

Once you have an application that is a discovery client, you can use it to discover service instances from the Eureka Server. One way to do so is to use the native `com.netflix.discovery.EurekaClient` (as opposed to the Spring Cloud `DiscoveryClient`), as shown in the following example:

```
@Autowired
private EurekaClient discoveryClient;

public String serviceUrl() {
    InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false);
    return instance.getHomePageUrl();
}
```

Do not use the `EurekaClient` in a `@PostConstruct` method or in a `@Scheduled` method (or anywhere where the `ApplicationContext` might not be started yet). It is initialized in a `SmartLifecycle` (with `phase=0`), so the earliest you can rely on it being available is in another `SmartLifecycle` with a higher phase.

129.8.1. EurekaClient without Jersey

By default, EurekaClient uses Jersey for HTTP communication. If you wish to avoid dependencies from Jersey, you can exclude it from your dependencies. Spring Cloud auto-configures a transport client based on Spring `RestTemplate`. The following example shows Jersey being excluded:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-client</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-core</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.sun.jersey.contribs</groupId>
            <artifactId>jersey-apache-client4</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

129.9. Alternatives to the Native Netflix EurekaClient

You need not use the raw Netflix `EurekaClient`. Also, it is usually more convenient to use it behind a wrapper of some sort. Spring Cloud has support for Feign (a REST client builder) and Spring `RestTemplate` through the logical Eureka service identifiers (VIPs) instead of physical URLs. To configure Ribbon with a fixed list of physical servers, you can set `<client>.ribbon.listOfServers` to a comma-separated list of physical addresses (or hostnames), where `<client>` is the ID of the client.

You can also use the `org.springframework.cloud.client.discovery.DiscoveryClient`, which provides a simple API (not specific to Netflix) for discovery clients, as shown in the following example:

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
    }
    return null;
}
```

129.10. Why Is It so Slow to Register a Service?

Being an instance also involves a periodic heartbeat to the registry (through the client's `serviceUrl`) with a default duration of 30 seconds. A service is not available for discovery by clients until the instance, the server, and the client all have the same metadata in their local cache (so it could take 3 heartbeats). You can change the period by setting `eureka.instance.leaseRenewalIntervalInSeconds`. Setting it to a value of less than 30 speeds up the process of getting clients connected to other services. In production, it is probably better to stick with the default, because of internal computations in the server that make assumptions about the lease renewal period.

129.11. Zones

If you have deployed Eureka clients to multiple zones, you may prefer that those clients use services within the same zone before trying services in another zone. To set that up, you need to configure your Eureka clients correctly.

First, you need to make sure you have Eureka servers deployed to each zone and that they are peers of each other. See the section on zones and regions for more information.

Next, you need to tell Eureka which zone your service is in. You can do so by using the `metadataMap` property. For example, if `service 1` is deployed to both `zone 1` and `zone 2`, you need to set the following Eureka properties in `service 1`:

Service 1 in Zone 1

```
eureka.instance.metadataMap.zone = zone1
eureka.client.preferSameZoneEureka = true
```

Service 1 in Zone 2

```
eureka.instance.metadataMap.zone = zone2
eureka.client.preferSameZoneEureka = true
```

129.12. Refreshing Eureka Clients

By default, the `EurekaClient` bean is refreshable, meaning the Eureka client properties can be changed and refreshed. When a refresh occurs clients will be unregistered from the Eureka server and there might be a brief moment of time where all instance of a given service are not available. One way to eliminate this from happening is to disable the ability to refresh Eureka clients. To do this set `eureka.client.refresh.enable=false`.

129.13. Using Eureka with Spring Cloud LoadBalancer

We offer support for the Spring Cloud LoadBalancer `ZonePreferenceServiceInstanceListSupplier`. The `zone` value from the Eureka instance metadata (`eureka.instance.metadataMap.zone`) is used for setting the value of `spring-clod-loadbalancer-zone` property that is used to filter service instances by zone.

If that is missing and if the `spring.cloud.loadbalancer.eureka.approximateZoneFromHostname` flag is set to `true`, it can use the domain name from the server hostname as a proxy for the zone.

If there is no other source of zone data, then a guess is made, based on the client configuration (as opposed to the instance configuration). We take `eureka.client.availabilityZones`, which is a map from region name to a list of zones, and pull out the first zone for the instance's own region (that is, the `eureka.client.region`, which defaults to "us-east-1", for compatibility with native Netflix).

Chapter 130. Service Discovery: Eureka Server

This section describes how to set up a Eureka server.

130.1. How to Include Eureka Server

To include Eureka Server in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-netflix-eureka-server`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

If your project already uses Thymeleaf as its template engine, the Freemarker templates of the Eureka server may not be loaded correctly. In this case it is necessary to configure the template loader manually:

application.yml

```
spring:
  freemarker:
    template-loader-path: classpath:/templates/
    prefer-file-system-access: false
```

130.2. How to Run a Eureka Server

The following example shows a minimal Eureka server:

```
@SpringBootApplication
@EnableEurekaServer
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

The server has a home page with a UI and HTTP API endpoints for the normal Eureka functionality under `/eureka/*`.

The following links have some Eureka background reading: flux capacitor and google group discussion.

Due to Gradle's dependency resolution rules and the lack of a parent bom feature, depending on `spring-cloud-starter-netflix-eureka-server` can cause failures on application startup. To remedy this issue, add the Spring Boot Gradle plugin and import the Spring cloud starter parent bom as follows:

build.gradle

```
buildscript {
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-
plugin:{spring-boot-docs-version}")
  }
}

apply plugin: "spring-boot"

dependencyManagement {
  imports {
    mavenBom "org.springframework.cloud:spring-cloud-
dependencies:{spring-cloud-version}"
  }
}
```

130.3. High Availability, Zones and Regions

The Eureka server does not have a back end store, but the service instances in the registry all have to send heartbeats to keep their registrations up to date (so this can be done in memory). Clients also have an in-memory cache of Eureka registrations (so they do not have to go to the registry for every request to a service).

By default, every Eureka server is also a Eureka client and requires (at least one) service URL to locate a peer. If you do not provide it, the service runs and works, but it fills your logs with a lot of noise about not being able to register with the peer.

See also below for details of Ribbon support on the client side for Zones and Regions.

130.4. Standalone Mode

The combination of the two caches (client and server) and the heartbeats make a standalone Eureka server fairly resilient to failure, as long as there is some sort of monitor or elastic runtime (such as Cloud Foundry) keeping it alive. In standalone mode, you might prefer to switch off the client side behavior so that it does not keep trying and failing to reach its peers. The following example shows how to switch off the client-side behavior:

application.yml (Standalone Eureka Server)

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

Notice that the `serviceUrl` is pointing to the same host as the local instance.

130.5. Peer Awareness

Eureka can be made even more resilient and available by running multiple instances and asking them to register with each other. In fact, this is the default behavior, so all you need to do to make it work is add a valid `serviceUrl` to a peer, as shown in the following example:

application.yml (Two Peer Aware Eureka Servers)

```
---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: https://peer2/eureka/

---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: https://peer1/eureka/
```

In the preceding example, we have a YAML file that can be used to run the same server on two hosts (`peer1` and `peer2`) by running it in different Spring profiles. You could use this configuration to test the peer awareness on a single host (there is not much value in doing that in production) by manipulating `/etc/hosts` to resolve the host names. In fact, the `eureka.instance.hostname` is not needed if you are running on a machine that knows its own hostname (by default, it is looked up by

using `java.net.InetAddress`).

You can add multiple peers to a system, and, as long as they are all connected to each other by at least one edge, they synchronize the registrations amongst themselves. If the peers are physically separated (inside a data center or between multiple data centers), then the system can, in principle, survive "split-brain" type failures. You can add multiple peers to a system, and as long as they are all directly connected to each other, they will synchronize the registrations amongst themselves.

application.yml (Three Peer Aware Eureka Servers)

```
eureka:
  client:
    serviceUrl:
      defaultZone: https://peer1/eureka/,http://peer2/eureka/,http://peer3/eureka/

---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1

---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2

---
spring:
  profiles: peer3
eureka:
  instance:
    hostname: peer3
```

130.6. When to Prefer IP Address

In some cases, it is preferable for Eureka to advertise the IP addresses of services rather than the hostname. Set `eureka.instance.preferIpAddress` to `true` and, when the application registers with eureka, it uses its IP address rather than its hostname.

If the hostname cannot be determined by Java, then the IP address is sent to Eureka. Only explict way of setting the hostname is by setting `eureka.instance.hostname` property. You can set your hostname at the run-time by using an environment variable — for example, `eureka.instance.hostname=${HOST_NAME}`.

130.7. Securing The Eureka Server

You can secure your Eureka server simply by adding Spring Security to your server's classpath via `spring-boot-starter-security`. By default when Spring Security is on the classpath it will require that a valid CSRF token be sent with every request to the app. Eureka clients will not generally possess a valid cross site request forgery (CSRF) token you will need to disable this requirement for the `/eureka/**` endpoints. For example:

```
@EnableWebSecurity
class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().ignoringAntMatchers("/eureka/**");
        super.configure(http);
    }
}
```

For more information on CSRF see the Spring Security documentation.

A demo Eureka Server can be found in the Spring Cloud Samples repo.

130.8. Disabling Ribbon with Eureka Server and Client starters

`spring-cloud-starter-netflix-eureka-server` and `spring-cloud-starter-netflix-eureka-client` come along with a `spring-cloud-starter-netflix-ribbon`. Since Ribbon load-balancer is now in maintenance mode, we suggest switching to using the Spring Cloud LoadBalancer, also included in Eureka starters, instead.

In order to that, you can set the value of `spring.cloud.loadbalancer.ribbon.enabled` property to `false`.

You can then also exclude ribbon-related dependencies from Eureka starters in your build files, like so:

```
<dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework.cloud</groupId>
                    <artifactId>spring-cloud-starter-ribbon</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>com.netflix.ribbon</groupId>
                    <artifactId>ribbon-eureka</artifactId>
                </exclusion>
            </exclusions>
</dependency>
```

130.9. JDK 11 Support

The JAXB modules which the Eureka server depends upon were removed in JDK 11. If you intend to use JDK 11 when running a Eureka server you must include these dependencies in your POM or Gradle file.

```
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
</dependency>
```

Chapter 131. Circuit Breaker: Spring Cloud Circuit Breaker With Hystrix

131.1. Disabling Spring Cloud Circuit Breaker Hystrix

You can disable the auto-configuration by setting `spring.cloud.circuitbreaker.hystrix.enabled` to `false`.

131.2. Configuring Hystrix Circuit Breakers

131.2.1. Default Configuration

To provide a default configuration for all of your circuit breakers create a `Customize` bean that is passed a `HystrixCircuitBreakerFactory` or `ReactiveHystrixCircuitBreakerFactory`. The `configureDefault` method can be used to provide a default configuration.

```
@Bean
public Customizer<HystrixCircuitBreakerFactory> defaultConfig() {
    return factory -> factory.configureDefault(id -> HystrixCommand.Setter
            .withGroupKey(HystrixCommandGroupKey.Factory.asKey(id))
            .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
            .withExecutionTimeoutInMilliseconds(4000)));
}
```

Reactive Example

```
@Bean
public Customizer<ReactiveHystrixCircuitBreakerFactory> defaultConfig() {
    return factory -> factory.configureDefault(id ->
HystrixObservableCommand.Setter
            .withGroupKey(HystrixCommandGroupKey.Factory.asKey(id))
            .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                    .withExecutionTimeoutInMilliseconds(4000)));
}
```

131.2.2. Specific Circuit Breaker Configuration

Similarly to providing a default configuration, you can create a `Customize` bean this is passed a `HystrixCircuitBreakerFactory`

```
@Bean
public Customizer<HystrixCircuitBreakerFactory> customizer() {
    return factory -> factory.configure(builder -> builder.commandProperties(

HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(2000)),
"foo", "bar");
}
```

Reactive Example

```
@Bean
public Customizer<ReactiveHystrixCircuitBreakerFactory> customizer() {
    return factory -> factory.configure(builder -> builder.commandProperties(

HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(2000)),
"foo", "bar");
}
```

Chapter 132. Circuit Breaker: Hystrix Clients

Netflix has created a library called Hystrix that implements the circuit breaker pattern. In a microservice architecture, it is common to have multiple layers of service calls, as shown in the following example:

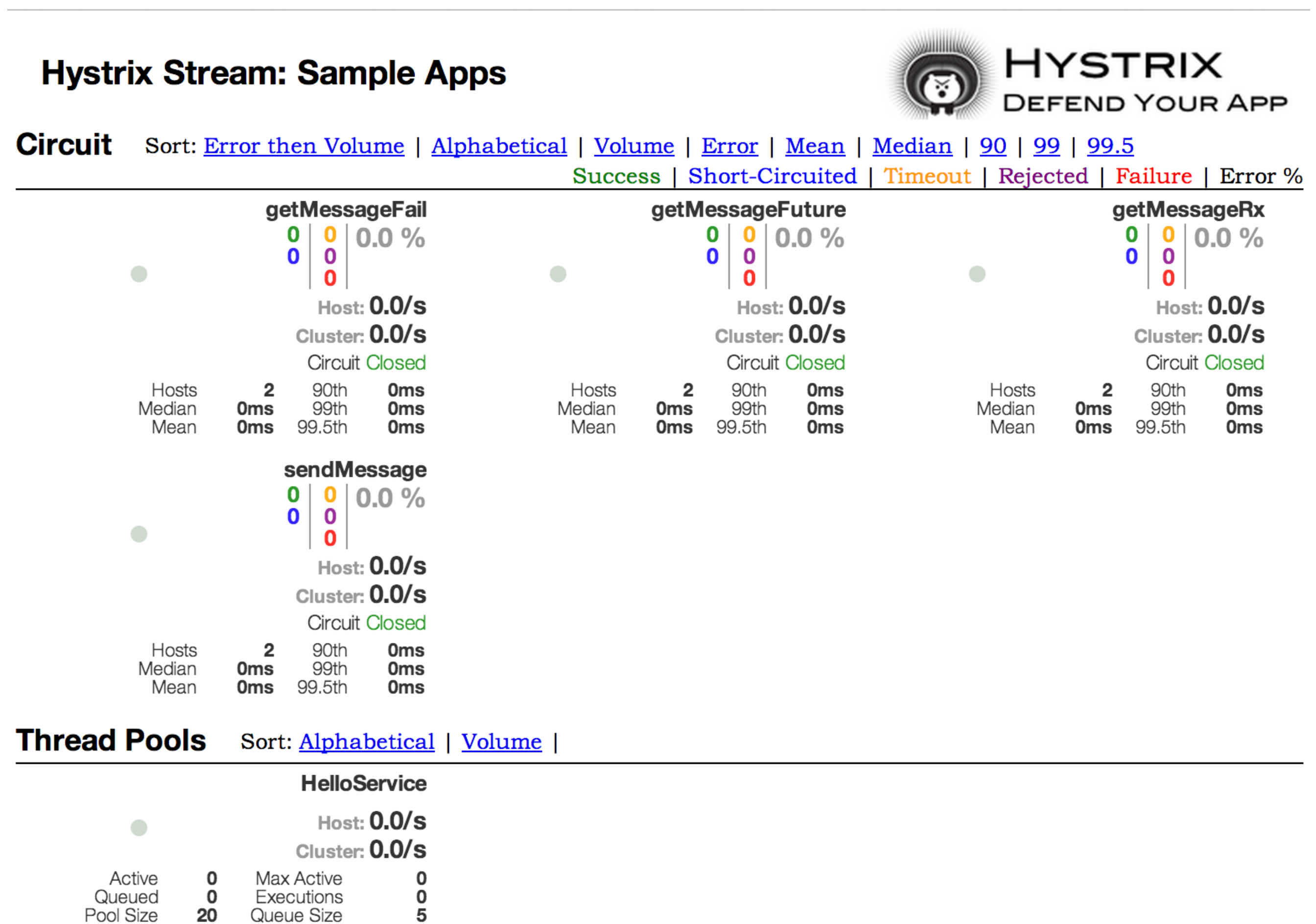


Figure 7. Microservice Graph

A service failure in the lower level of services can cause cascading failure all the way up to the user. When calls to a particular service exceed `circuitBreaker.requestVolumeThreshold` (default: 20 requests) and the failure percentage is greater than `circuitBreaker.errorThresholdPercentage` (default: >50%) in a rolling window defined by `metrics.rollingStats.timeInMilliseconds` (default: 10 seconds), the circuit opens and the call is not made. In cases of error and an open circuit, a fallback can be provided by the developer.

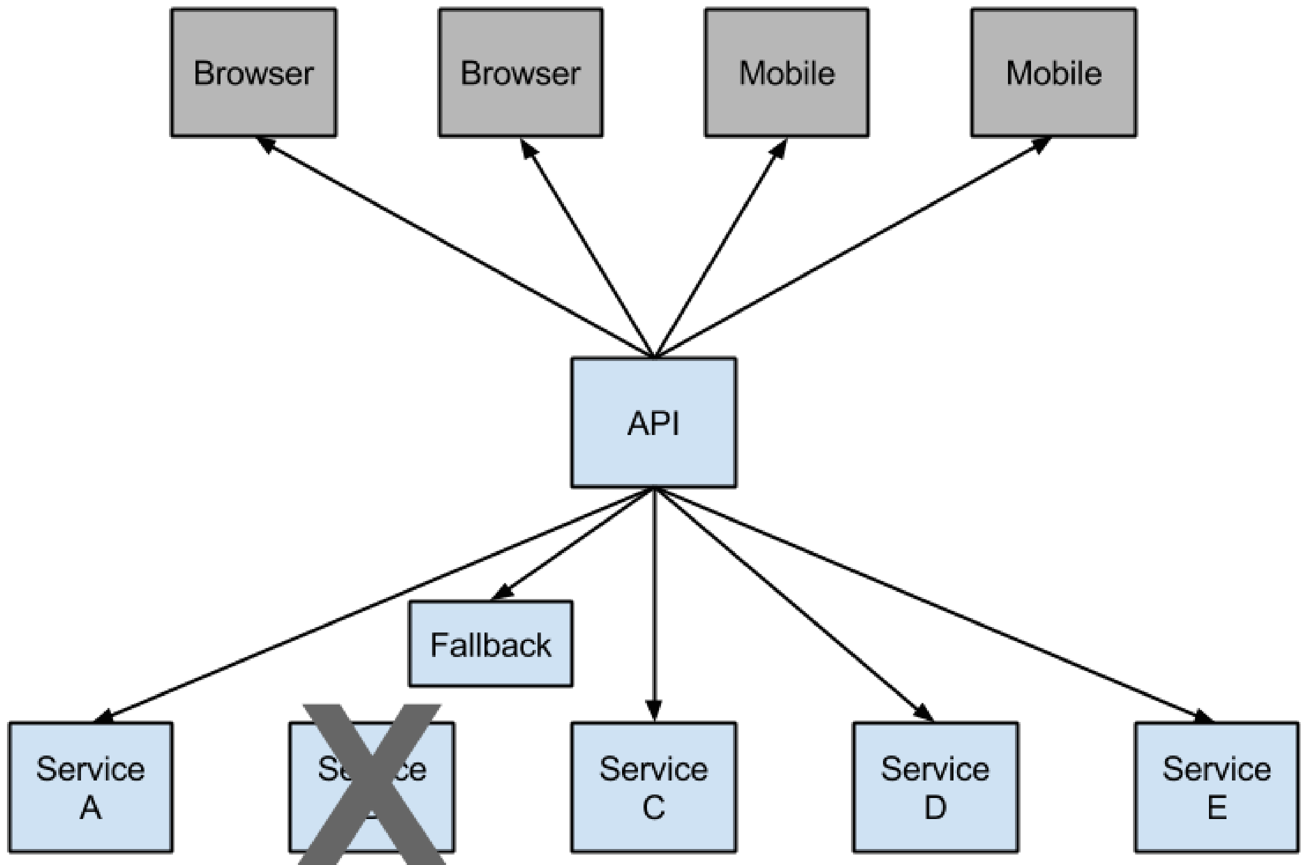


Figure 8. Hystrix fallback prevents cascading failures

Having an open circuit stops cascading failures and allows overwhelmed or failing services time to recover. The fallback can be another Hystrix protected call, static data, or a sensible empty value. Fallbacks may be chained so that the first fallback makes some other business call, which in turn falls back to static data.

132.1. How to Include Hystrix

To include Hystrix in your project, use the starter with a group ID of `org.springframework.cloud` and a artifact ID of `spring-cloud-starter-netflix-hystrix`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

The following example shows a minimal Eureka server with a Hystrix circuit breaker:

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}

@Component
public class StoreIntegration {

    @HystrixCommand(fallbackMethod = "defaultStores")
    public Object getStores(Map<String, Object> parameters) {
        //do stuff that might fail
    }

    public Object defaultStores(Map<String, Object> parameters) {
        return /* something useful */;
    }
}
```

The `@HystrixCommand` is provided by a Netflix contrib library called "javanica". Spring Cloud automatically wraps Spring beans with that annotation in a proxy that is connected to the Hystrix circuit breaker. The circuit breaker calculates when to open and close the circuit and what to do in case of a failure.

To configure the `@HystrixCommand` you can use the `commandProperties` attribute with a list of `@HystrixProperty` annotations. See here for more details. See the Hystrix wiki for details on the properties available.

132.2. Propagating the Security Context or Using Spring Scopes

If you want some thread local context to propagate into a `@HystrixCommand`, the default declaration does not work, because it executes the command in a thread pool (in case of timeouts). You can switch Hystrix to use the same thread as the caller through configuration or directly in the annotation, by asking it to use a different "Isolation Strategy". The following example demonstrates setting the thread in the annotation:

```
@HystrixCommand(fallbackMethod = "stubMyService",
    commandProperties = {
      @HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")
    }
)
...
```

The same thing applies if you are using `@SessionScope` or `@RequestScope`. If you encounter a runtime exception that says it cannot find the scoped context, you need to use the same thread.

You also have the option to set the `hystrix.shareSecurityContext` property to `true`. Doing so auto-configures a Hystrix concurrency strategy plugin hook to transfer the `SecurityContext` from your main thread to the one used by the Hystrix command. Hystrix does not let multiple Hystrix concurrency strategy be registered so an extension mechanism is available by declaring your own `HystrixConcurrencyStrategy` as a Spring bean. Spring Cloud looks for your implementation within the Spring context and wrap it inside its own plugin.

132.3. Health Indicator

The state of the connected circuit breakers are also exposed in the `/health` endpoint of the calling application, as shown in the following example:

```
{
    "hystrix": {
        "openCircuitBreakers": [
            "StoreIntegration::getStoresByLocationLink"
        ],
        "status": "CIRCUIT_OPEN"
    },
    "status": "UP"
}
```

132.4. Hystrix Metrics Stream

To enable the Hystrix metrics stream, include a dependency on `spring-boot-starter-actuator` and set `management.endpoints.web.exposure.include: hystrix.stream`. Doing so exposes the `/actuator/hystrix.stream` as a management endpoint, as shown in the following example:

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

Chapter 133. Circuit Breaker: Hystrix Dashboard

One of the main benefits of Hystrix is the set of metrics it gathers about each HystrixCommand. The Hystrix Dashboard displays the health of each circuit breaker in an efficient manner.

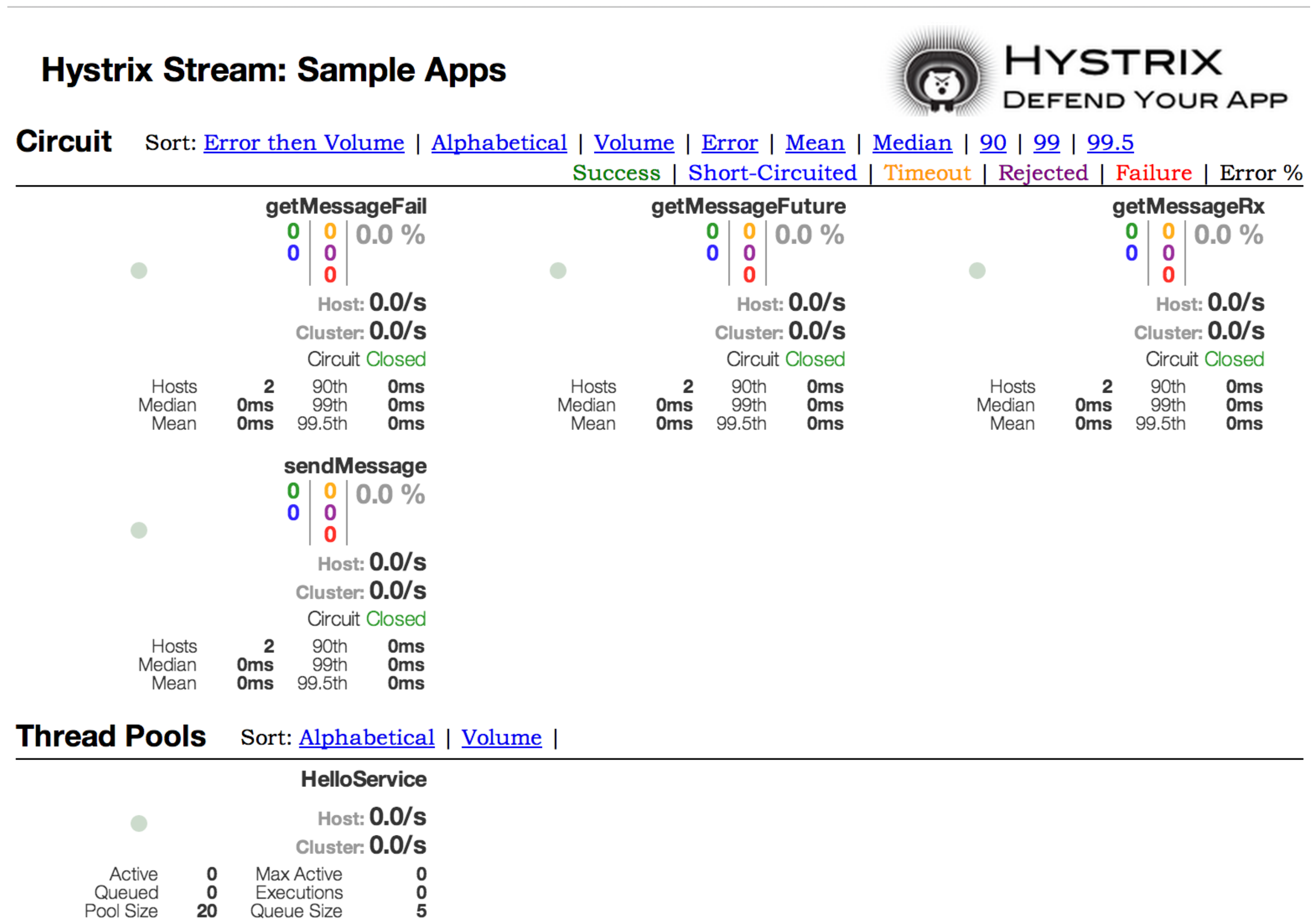


Figure 9. Hystrix Dashboard

Chapter 134. Hystrix Timeouts And Ribbon Clients

When using Hystrix commands that wrap Ribbon clients you want to make sure your Hystrix timeout is configured to be longer than the configured Ribbon timeout, including any potential retries that might be made. For example, if your Ribbon connection timeout is one second and the Ribbon client might retry the request three times, than your Hystrix timeout should be slightly more than three seconds.

134.1. How to Include the Hystrix Dashboard

To include the Hystrix Dashboard in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-netflix-hystrix-dashboard`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

To run the Hystrix Dashboard, annotate your Spring Boot main class with `@EnableHystrixDashboard`. Then visit `/hystrix` and point the dashboard to an individual instance's `/hystrix.stream` endpoint in a Hystrix client application.

When connecting to a `/hystrix.stream` endpoint that uses HTTPS, the certificate used by the server must be trusted by the JVM. If the certificate is not trusted, you must import the certificate into the JVM in order for the Hystrix Dashboard to make a successful connection to the stream endpoint.

In order to use the `/proxy.stream` endpoint you must configure a list of hosts to allow connections to. To set the list of allowed hosts use `hystrix.dashboard.proxyStreamAllowList`. You can use an Ant-style pattern in the host name to match against a wider range of host names.

134.2. Turbine

Looking at an individual instance's Hystrix data is not very useful in terms of the overall health of the system. Turbine is an application that aggregates all of the relevant `/hystrix.stream` endpoints into a combined `/turbine.stream` for use in the Hystrix Dashboard. Individual instances are located through Eureka. Running Turbine requires annotating your main class with the `@EnableTurbine` annotation (for example, by using spring-cloud-starter-netflix-turbine to set up the classpath). All of the documented configuration properties from the Turbine 1 wiki apply. The only difference is that the `turbine.instanceUrlSuffix` does not need the port prepended, as this is handled automatically unless `turbine.instanceInsertPort=false`.

By default, Turbine looks for the `/hystrix.stream` endpoint on a registered instance by looking up its `hostName` and `port` entries in Eureka and then appending `/hystrix.stream` to it. If the instance's metadata contains `management.port`, it is used instead of the `port` value for the `/hystrix.stream` endpoint. By default, the metadata entry called `management.port` is equal to the `management.port` configuration property. It can be overridden though with following configuration:

```
eureka:
  instance:
    metadata-map:
      management.port: ${management.port:8081}
```

The `turbine.appConfig` configuration key is a list of Eureka serviceIds that turbine uses to lookup instances. The turbine stream is then used in the Hystrix dashboard with a URL similar to the following:

`my.turbine.server:8080/turbine.stream?cluster=CLUSTERNAME`

The cluster parameter can be omitted if the name is `default`. The `cluster` parameter must match an entry in `turbine.aggregator.clusterConfig`. Values returned from Eureka are upper-case. Consequently, the following example works if there is an application called `customers` registered with Eureka:

```
turbine:
  aggregator:
    clusterConfig: CUSTOMERS
  appConfig: customers
```

If you need to customize which cluster names should be used by Turbine (because you do not want to store cluster names in `turbine.aggregator.clusterConfig` configuration), provide a bean of type `TurbineClustersProvider`.

The `clusterName` can be customized by a SPEL expression in `turbine.clusterNameExpression` with root as an instance of `InstanceInfo`. The default value is `appName`, which means that the Eureka `serviceId` becomes the cluster key (that is, the `InstanceInfo` for customers has an `appName` of `CUSTOMERS`). A different example is `turbine.clusterNameExpression=aSGName`, which gets the cluster name from the AWS ASG name. The following listing shows another example:

```
turbine:
  aggregator:
    clusterConfig: SYSTEM,USER
  appConfig: customers,stores,ui,admin
  clusterNameExpression: metadata['cluster']
```

In the preceding example, the cluster name from four services is pulled from their metadata map and is expected to have values that include `SYSTEM` and `USER`.

To use the “default” cluster for all apps, you need a string literal expression (with single quotes and escaped with double quotes if it is in YAML as well):

```
turbine:
  appConfig: customers,stores
  clusterNameExpression: "'default'"
```

Spring Cloud provides a `spring-cloud-starter-netflix-turbine` that has all the dependencies you need to get a Turbine server running. To add Turbine, create a Spring Boot application and annotate it with `@EnableTurbine`.

By default, Spring Cloud lets Turbine use the host and port to allow multiple processes per host, per cluster. If you want the native Netflix behavior built into Turbine to *not* allow multiple processes per host, per cluster (the key to the instance ID is the hostname), set `turbine.combineHostPort=false`.

134.2.1. Clusters Endpoint

In some situations it might be useful for other applications to know what custers have been configured in Turbine. To support this you can use the `/clusters` endpoint which will return a JSON array of all the configured clusters.

GET /clusters

```
[
  {
    "name": "RACES",
    "link": "http://localhost:8383/turbine.stream?cluster=RACES"
  },
  {
    "name": "WEB",
    "link": "http://localhost:8383/turbine.stream?cluster=WEB"
  }
]
```

This endpoint can be disabled by setting `turbine.endpoints.clusters.enabled` to `false`.

134.3. Turbine Stream

In some environments (such as in a PaaS setting), the classic Turbine model of pulling metrics from all the distributed Hystrix commands does not work. In that case, you might want to have your Hystrix commands push metrics to Turbine. Spring Cloud enables that with messaging. To do so on the client, add a dependency to `spring-cloud-netflix-hystrix-stream` and the `spring-cloud-starter-stream-*` of your choice. See the Spring Cloud Stream documentation for details on the brokers and how to configure the client credentials. It should work out of the box for a local broker.

On the server side, create a Spring Boot application and annotate it with `@EnableTurbineStream`. The

Turbine Stream server requires the use of Spring Webflux, therefore `spring-boot-starter-webflux` needs to be included in your project. By default `spring-boot-starter-webflux` is included when adding `spring-cloud-starter-netflix-turbine-stream` to your application.

You can then point the Hystrix Dashboard to the Turbine Stream Server instead of individual Hystrix streams. If Turbine Stream is running on port 8989 on myhost, then put `myhost:8989` in the stream input field in the Hystrix Dashboard. Circuits are prefixed by their respective `serviceId`, followed by a dot (`.`), and then the circuit name.

Spring Cloud provides a `spring-cloud-starter-netflix-turbine-stream` that has all the dependencies you need to get a Turbine Stream server running. You can then add the Stream binder of your choice — such as `spring-cloud-starter-stream-rabbit`.

Turbine Stream server also supports the `cluster` parameter. Unlike Turbine server, Turbine Stream uses eureka serviceIds as cluster names and these are not configurable.

If Turbine Stream server is running on port 8989 on `my.turbine.server` and you have two eureka serviceIds `customers` and `products` in your environment, the following URLs will be available on your Turbine Stream server. `default` and empty cluster name will provide all metrics that Turbine Stream server receives.

```
https://my.turbine.sever:8989/turbine.stream?cluster=customers
https://my.turbine.sever:8989/turbine.stream?cluster=products
https://my.turbine.sever:8989/turbine.stream?cluster=default
https://my.turbine.sever:8989/turbine.stream
```

So, you can use eureka serviceIds as cluster names for your Turbine dashboard (or any compatible dashboard). You don’t need to configure any properties like `turbine.appConfig`, `turbine.clusterNameExpression` and `turbine.aggregator.clusterConfig` for your Turbine Stream server.

Turbine Stream server gathers all metrics from the configured input channel with Spring Cloud Stream. It means that it doesn’t gather Hystrix metrics actively from each instance. It just can provide metrics that were already gathered into the input channel by each instance.

Chapter 135. Client Side Load Balancer: Ribbon

Ribbon is a client-side load balancer that gives you a lot of control over the behavior of HTTP and TCP clients. Feign already uses Ribbon, so, if you use `@FeignClient`, this section also applies.

A central concept in Ribbon is that of the named client. Each load balancer is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer (for example, by using the `@FeignClient` annotation). On demand, Spring Cloud creates a new ensemble as an `ApplicationContext` for each named client by using `RibbonClientConfiguration`. This contains (amongst other things) an `ILoadBalancer`, a `RestClient`, and a `ServerListFilter`.

135.1. How to Include Ribbon

To include Ribbon in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-netflix-ribbon`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

135.2. Customizing the Ribbon Client

You can configure some bits of a Ribbon client by using external properties in `<client>.ribbon.*`, which is similar to using the Netflix APIs natively, except that you can use Spring Boot configuration files. The native options can be inspected as static fields in `CommonClientConfigKey` (part of ribbon-core).

Spring Cloud also lets you take full control of the client by declaring additional configuration (on top of the `RibbonClientConfiguration`) using `@RibbonClient`, as shown in the following example:

```
@Configuration
@RibbonClient(name = "custom", configuration = CustomConfiguration.class)
public class TestConfiguration {
}
```

In this case, the client is composed from the components already in `RibbonClientConfiguration`, together with any in `CustomConfiguration` (where the latter generally overrides the former).

The `CustomConfiguration` class must be a `@Configuration` class, but take care that it is not in a `@ComponentScan` for the main application context. Otherwise, it is shared by all the `@RibbonClients`. If you use `@ComponentScan` (or `@SpringBootApplication`), you need to take steps to avoid it being included (for instance, you can put it in a separate, non-overlapping package or specify the packages to scan explicitly in the `@ComponentScan`).

The following table shows the beans that Spring Cloud Netflix provides by default for Ribbon:

Bean Type	Bean Name	Class Name
`IClientConfig`	`ribbonClientConfig`	`DefaultClientConfigImpl`
`IRule`	`ribbonRule`	`ZoneAvoidanceRule`
`IPing`	`ribbonPing`	`DummyPing`
`ServerList<Server>`	`ribbonServerList`	`ConfigurationBasedServerList`
`ServerListFilter<Server>`	`ribbonServerListFilter`	`ZonePreferenceServerListFilter`
`ILoadBalancer`	`ribbonLoadBalancer`	`ZoneAwareLoadBalancer`
`ServerListUpdater`	`ribbonServerListUpdater`	`PollingServerListUpdater`

Creating a bean of one of those type and placing it in a `@RibbonClient` configuration (such as `FooConfiguration` above) lets you override each one of the beans described, as shown in the following example:

```
@Configuration(proxyBeanMethods = false)
protected static class FooConfiguration {

    @Bean
    public ZonePreferenceServerListFilter serverListFilter() {
        ZonePreferenceServerListFilter filter = new ZonePreferenceServerListFilter();
        filter.setZone("myTestZone");
        return filter;
    }

    @Bean
    public IPing ribbonPing() {
        return new PingUrl();
    }

}
```

The include statement in the preceding example replaces `NoOpPing` with `PingUrl` and provides a custom `serverListFilter`.

135.3. Customizing the Default for All Ribbon Clients

A default configuration can be provided for all Ribbon Clients by using the `@RibbonClients` annotation and registering a default configuration, as shown in the following example:

```java
@RibbonClients(defaultConfiguration = DefaultRibbonConfig.class)
public class RibbonClientDefaultConfigurationTestsConfig {

    public static class BazServiceList extends ConfigurationBasedServerList {

        public BazServiceList(IClientConfig config) {
            super.initWithNiwsConfig(config);
        }

    }

}

@Configuration(proxyBeanMethods = false)
class DefaultRibbonConfig {

    @Bean
    public IRule ribbonRule() {
        return new BestAvailableRule();
    }

    @Bean
    public IPing ribbonPing() {
        return new PingUrl();
    }

    @Bean
    public ServerList<Server> ribbonServerList(IClientConfig config) {
        return new RibbonClientDefaultConfigurationTestsConfig.BazServiceList(config);
    }

    @Bean
    public ServerListSubsetFilter serverListFilter() {
        ServerListSubsetFilter filter = new ServerListSubsetFilter();
        return filter;
    }

}
```

135.4. Customizing the Ribbon Client by Setting Properties

Starting with version 1.2.0, Spring Cloud Netflix now supports customizing Ribbon clients by setting properties to be compatible with the Ribbon documentation.

This lets you change behavior at start up time in different environments.

The following list shows the supported properties>:

- `<clientName>.ribbon.NFLoadBalancerClassName`: Should implement `ILoadBalancer`
- `<clientName>.ribbon.NFLoadBalancerRuleClassName`: Should implement `IRule`
- `<clientName>.ribbon.NFLoadBalancerPingClassName`: Should implement `IPing`
- `<clientName>.ribbon.NIWSServerListClassName`: Should implement `ServerList`
- `<clientName>.ribbon.NIWSServerListFilterClassName`: Should implement `ServerListFilter`

Classes defined in these properties have precedence over beans defined by using `@RibbonClient(configuration=MyRibbonConfig.class)` and the defaults provided by Spring Cloud Netflix.

To set the `IRule` for a service name called `users`, you could set the following properties:

application.yml

```
users:
  ribbon:
    NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.WeightedResponseTimeRule
```

See the Ribbon documentation for implementations provided by Ribbon.

135.5. Using Ribbon with Eureka

When Eureka is used in conjunction with Ribbon (that is, both are on the classpath), the `ribbonServerList` is overridden with an extension of `DiscoveryEnabledNIWSServerList`, which populates the list of servers from Eureka. It also replaces the `IPing` interface with `NIWSDiscoveryPing`, which delegates to Eureka to determine if a server is up. The `ServerList` that is installed by default is a `DomainExtractingServerList`. Its purpose is to make metadata available to the load balancer without using AWS AMI metadata (which is what Netflix relies on). By default, the server list is constructed with “zone” information, as provided in the instance metadata (so, on the remote clients, set `eureka.instance.metadataMap.zone`). If that is missing and if the `approximateZoneFromHostname` flag is set, it can use the domain name from the server hostname as a proxy for the zone. Once the zone information is available, it can be used in a `ServerListFilter`. By default, it is used to locate a server in the same zone as the client, because the default is a `ZonePreferenceServerListFilter`. By default, the zone of the client is determined in the same way as the remote instances (that is, through `eureka.instance.metadataMap.zone`).

The orthodox “archaius” way to set the client zone is through a configuration property called "@zone". If it is available, Spring Cloud uses that in preference to all other settings (note that the key must be quoted in YAML configuration).

If there is no other source of zone data, then a guess is made, based on the client configuration (as opposed to the instance configuration). We take `eureka.client.availabilityZones`, which is a map from region name to a list of zones, and pull out the first zone for the instance's own region (that is, the `eureka.client.region`, which defaults to "us-east-1", for compatibility with native Netflix).

135.6. Example: How to Use Ribbon Without Eureka

Eureka is a convenient way to abstract the discovery of remote servers so that you do not have to hard code their URLs in clients. However, if you prefer not to use Eureka, Ribbon and Feign also work. Suppose you have declared a `@RibbonClient` for "stores", and Eureka is not in use (and not even on the classpath). The Ribbon client defaults to a configured server list. You can supply the configuration as follows:

application.yml

```
stores:
  ribbon:
    listOfServers: example.com,google.com
```

135.7. Example: Disable Eureka Use in Ribbon

Setting the `ribbon.eureka.enabled` property to `false` explicitly disables the use of Eureka in Ribbon, as shown in the following example:

application.yml

```
ribbon:
  eureka:
   enabled: false
```

135.8. Using the Ribbon API Directly

You can also use the `LoadBalancerClient` directly, as shown in the following example:

```
public class MyClass {
    @Autowired
    private LoadBalancerClient loadBalancer;

    public void doStuff() {
        ServiceInstance instance = loadBalancer.choose("stores");
        URI storesUri = URI.create(String.format("https://%s:%s", instance.getHost(),
instance.getPort()));
        // ... do something with the URI
    }
}
```

135.9. Caching of Ribbon Configuration

Each Ribbon named client has a corresponding child application Context that Spring Cloud maintains. This application context is lazily loaded on the first request to the named client. This lazy loading behavior can be changed to instead eagerly load these child application contexts at startup, by specifying the names of the Ribbon clients, as shown in the following example:

application.yml

```
ribbon:
  eager-load:
    enabled: true
    clients: client1, client2, client3
```

135.10. How to Configure Hystrix Thread Pools

If you change `zuul.ribbonIsolationStrategy` to `THREAD`, the thread isolation strategy for Hystrix is used for all routes. In that case, the `HystrixThreadPoolKey` is set to `RibbonCommand` as the default. It means that HystrixCommands for all routes are executed in the same Hystrix thread pool. This behavior can be changed with the following configuration:

application.yml

```
zuul:
  threadPool:
    useSeparateThreadPools: true
```

The preceding example results in HystrixCommands being executed in the Hystrix thread pool for each route.

In this case, the default `HystrixThreadPoolKey` is the same as the service ID for each route. To add a prefix to `HystrixThreadPoolKey`, set `zuul.threadPool.threadPoolKeyPrefix` to the value that you want to add, as shown in the following example:

application.yml

```
zuul:
  threadPool:
    useSeparateThreadPools: true
    threadPoolKeyPrefix: zuulgw
```

135.11. How to Provide a Key to Ribbon's `IRule`

If you need to provide your own `IRule` implementation to handle a special routing requirement like a "canary" test, pass some information to the `choose` method of `IRule`.

com.netflix.loadbalancer.IRule.java

```
public interface IRule{
    public Server choose(Object key);
         :
```

You can provide some information that is used by your `IRule` implementation to choose a target server, as shown in the following example:

```
RequestContext.getCurrentContext()
              .set(FilterConstants.LOAD_BALANCER_KEY, "canary-test");
```

If you put any object into the `RequestContext` with a key of `FilterConstants.LOAD_BALANCER_KEY`, it is passed to the `choose` method of the `IRule` implementation. The code shown in the preceding example must be executed before `RibbonRoutingFilter` is executed. Zuul's pre filter is the best place to do that. You can access HTTP headers and query parameters through the `RequestContext` in pre filter, so it can be used to determine the `LOAD_BALANCER_KEY` that is passed to Ribbon. If you do not put any value with `LOAD_BALANCER_KEY` in `RequestContext`, null is passed as a parameter of the `choose` method.

Chapter 136. External Configuration: Archaius

Archaius is the Netflix client-side configuration library. It is the library used by all of the Netflix OSS components for configuration. Archaius is an extension of the Apache Commons Configuration project. It allows updates to configuration by either polling a source for changes or by letting a source push changes to the client. Archaius uses Dynamic<Type>Property classes as handles to properties, as shown in the following example:

Archaius Example

```
class ArchaiusTest {
    DynamicStringProperty myprop = DynamicPropertyFactory
            .getInstance()
            .getStringProperty("my.prop");

    void doSomething() {
        OtherClass.someMethod(myprop.get());
    }
}
```

Archaius has its own set of configuration files and loading priorities. Spring applications should generally not use Archaius directly, but the need to configure the Netflix tools natively remains. Spring Cloud has a Spring Environment Bridge so that Archaius can read properties from the Spring Environment. This bridge allows Spring Boot projects to use the normal configuration toolchain while letting them configure the Netflix tools as documented (for the most part).

Chapter 137. Router and Filter: Zuul

Routing is an integral part of a microservice architecture. For example, `/` may be mapped to your web application, `/api/users` is mapped to the user service and `/api/shop` is mapped to the shop service. Zuul is a JVM-based router and server-side load balancer from Netflix.

Netflix uses Zuul for the following:

- Authentication
- Insights
- Stress Testing
- Canary Testing
- Dynamic Routing
- Service Migration
- Load Shedding
- Security
- Static Response handling
- Active/Active traffic management

Zuul's rule engine lets rules and filters be written in essentially any JVM language, with built-in support for Java and Groovy.

The configuration property `zuul.max.host.connections` has been replaced by two new properties, `zuul.host.maxTotalConnections` and `zuul.host.maxPerRouteConnections`, which default to 200 and 20 respectively.

The default Hystrix isolation pattern (`ExecutionIsolationStrategy`) for all routes is `SEMAPHORE`. `zuul.ribbonIsolationStrategy` can be changed to `THREAD` if that isolation pattern is preferred.

137.1. How to Include Zuul

To include Zuul in your project, use the starter with a group ID of `org.springframework.cloud` and a artifact ID of `spring-cloud-starter-netflix-zuul`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

137.2. Embedded Zuul Reverse Proxy

Spring Cloud has created an embedded Zuul proxy to ease the development of a common use case where a UI application wants to make proxy calls to one or more back end services. This feature is useful for a user interface to proxy to the back end services it requires, avoiding the need to manage CORS and authentication concerns independently for all the back ends.

To enable it, annotate a Spring Boot main class with `@EnableZuulProxy`. Doing so causes local calls to be forwarded to the appropriate service. By convention, a service with an ID of `users` receives requests from the proxy located at `/users` (with the prefix stripped). The proxy uses Ribbon to locate an instance to which to forward through discovery. All requests are executed in a hystrix command, so failures appear in Hystrix metrics. Once the circuit is open, the proxy does not try to contact the service.

the Zuul starter does not include a discovery client, so, for routes based on service IDs, you need to provide one of those on the classpath as well (Eureka is one choice).

To skip having a service automatically added, set `zuul.ignored-services` to a list of service ID patterns. If a service matches a pattern that is ignored but is also included in the explicitly configured routes map, it is unignored, as shown in the following example:

application.yml

```
zuul:
 ignoredServices: '*'
 routes:
   users: /myusers/**
```

In the preceding example, all services are ignored, **except** for `users`.

To augment or change the proxy routes, you can add external configuration, as follows:

application.yml

```
zuul:
 routes:
   users: /myusers/**
```

The preceding example means that HTTP calls to `/myusers` get forwarded to the `users` service (for example `/myusers/101` is forwarded to `/101`).

To get more fine-grained control over a route, you can specify the path and the serviceId independently, as follows:

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
     serviceId: users_service
```

The preceding example means that HTTP calls to `/myusers` get forwarded to the `users_service` service. The route must have a `path` that can be specified as an ant-style pattern, so `/myusers/*` only

matches one level, but `/myusers/**` matches hierarchically.

The location of the back end can be specified as either a `serviceId` (for a service from discovery) or a `url` (for a physical location), as shown in the following example:

application.yml

```
 zuul:
  routes:
    users:
      path: /myusers/**
      url: https://example.com/users_service
```

These simple url-routes do not get executed as a `HystrixCommand`, nor do they load-balance multiple URLs with Ribbon. To achieve those goals, you can specify a `serviceId` with a static list of servers, as follows:

application.yml

```
zuul:
  routes:
    echo:
      path: /myusers/**
      serviceId: myusers-service
      stripPrefix: true

hystrix:
  command:
    myusers-service:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: ...

myusers-service:
  ribbon:
    NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList
    listOfServers: https://example1.com,http://example2.com
    ConnectTimeout: 1000
    ReadTimeout: 3000
    MaxTotalHttpConnections: 500
    MaxConnectionsPerHost: 100
```

Another method is specifiying a service-route and configuring a Ribbon client for the `serviceId` (doing so requires disabling Eureka support in Ribbon — see above for more information), as shown in the following example:

application.yml

```
zuul:
  routes:
    users:
      path: /myusers/**
      serviceId: users

ribbon:
  eureka:
    enabled: false

users:
  ribbon:
    listOfServers: example.com,google.com
```

You can provide a convention between `serviceId` and routes by using `regexmapper`. It uses regular-expression named groups to extract variables from `serviceId` and inject them into a route pattern, as shown in the following example:

ApplicationConfiguration.java

```
@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
    return new PatternServiceRouteMapper(
        "(?<name>^.+)-(?<version>v.+$)",
        "${version}/${name}");
}
```

The preceding example means that a `serviceId` of `myusers-v1` is mapped to route `/v1/myusers/**`. Any regular expression is accepted, but all named groups must be present in both `servicePattern` and `routePattern`. If `servicePattern` does not match a `serviceId`, the default behavior is used. In the preceding example, a `serviceId` of `myusers` is mapped to the "/myusers/**" route (with no version detected). This feature is disabled by default and only applies to discovered services.

To add a prefix to all mappings, set `zuul.prefix` to a value, such as `/api`. By default, the proxy prefix is stripped from the request before the request is forwarded by (you can switch this behavior off with `zuul.stripPrefix=false`). You can also switch off the stripping of the service-specific prefix from individual routes, as shown in the following example:

application.yml

```
 zuul:
  routes:
    users:
      path: /myusers/**
      stripPrefix: false
```

`zuul.stripPrefix` only applies to the prefix set in `zuul.prefix`. It does not have any effect on prefixes defined within a given route's `path`.

In the preceding example, requests to `/myusers/101` are forwarded to `/myusers/101` on the `users` service.

The `zuul.routes` entries actually bind to an object of type `ZuulProperties`. If you look at the properties of that object, you can see that it also has a `retryable` flag. Set that flag to `true` to have the Ribbon client automatically retry failed requests. You can also set that flag to `true` when you need to modify the parameters of the retry operations that use the Ribbon client configuration.

By default, the `X-Forwarded-Host` header is added to the forwarded requests. To turn it off, set `zuul.addProxyHeaders = false`. By default, the prefix path is stripped, and the request to the back end picks up a `X-Forwarded-Prefix` header (`/myusers` in the examples shown earlier).

If you set a default route (`/`), an application with `@EnableZuulProxy` could act as a standalone server. For example, `zuul.route.home: /` would route all traffic ("/**") to the "home" service.

If more fine-grained ignoring is needed, you can specify specific patterns to ignore. These patterns are evaluated at the start of the route location process, which means prefixes should be included in the pattern to warrant a match. Ignored patterns span all services and supersede any other route specification. The following example shows how to create ignored patterns:

application.yml

```
zuul:
 ignoredPatterns: /**/admin/**
 routes:
   users: /myusers/**
```

The preceding example means that all calls (such as `/myusers/101`) are forwarded to `/101` on the `users` service. However, calls including `/admin/` do not resolve.

If you need your routes to have their order preserved, you need to use a YAML file, as the ordering is lost when using a properties file. The following example shows such a YAML file:

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
   legacy:
     path: /**
```

If you were to use a properties file, the `legacy` path might end up in front of the `users` path, rendering the `users` path unreachable.

137.3. Zuul Http Client

The default HTTP client used by Zuul is now backed by the Apache HTTP Client instead of the deprecated Ribbon `RestClient`. To use `RestClient` or `okhttp3.OkHttpClient`, set `ribbon.restclient.enabled=true` or `ribbon.okhttp.enabled=true`, respectively. If you would like to customize the Apache HTTP client or the OK HTTP client, provide a bean of type `CloseableHttpClient` or `OkHttpClient`.

137.4. Cookies and Sensitive Headers

You can share headers between services in the same system, but you probably do not want sensitive headers leaking downstream into external servers. You can specify a list of ignored headers as part of the route configuration. Cookies play a special role, because they have well defined semantics in browsers, and they are always to be treated as sensitive. If the consumer of your proxy is a browser, then cookies for downstream services also cause problems for the user, because they all get jumbled up together (all downstream services look like they come from the same place).

If you are careful with the design of your services, (for example, if only one of the downstream services sets cookies), you might be able to let them flow from the back end all the way up to the caller. Also, if your proxy sets cookies and all your back-end services are part of the same system, it can be natural to simply share them (and, for instance, use Spring Session to link them up to some shared state). Other than that, any cookies that get set by downstream services are likely to be not useful to the caller, so it is recommended that you make (at least) `Set-Cookie` and `Cookie` into sensitive headers for routes that are not part of your domain. Even for routes that are part of your domain, try to think carefully about what it means before letting cookies flow between them and the proxy.

The sensitive headers can be configured as a comma-separated list per route, as shown in the following example:

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders: Cookie,Set-Cookie,Authorization
     url: https://downstream
```

This is the default value for `sensitiveHeaders`, so you need not set it unless you want it to be different. This is new in Spring Cloud Netflix 1.1 (in 1.0, the user had no control over headers, and all cookies flowed in both directions).

The `sensitiveHeaders` are a blacklist, and the default is not empty. Consequently, to make Zuul send all headers (except the `ignored` ones), you must explicitly set it to the empty list. Doing so is necessary if you want to pass cookie or authorization headers to your back end. The following example shows how to use `sensitiveHeaders`:

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders:
     url: https://downstream
```

You can also set sensitive headers, by setting `zuul.sensitiveHeaders`. If `sensitiveHeaders` is set on a route, it overrides the global `sensitiveHeaders` setting.

137.5. Ignored Headers

In addition to the route-sensitive headers, you can set a global value called `zuul.ignoredHeaders` for values (both request and response) that should be discarded during interactions with downstream services. By default, if Spring Security is not on the classpath, these are empty. Otherwise, they are initialized to a set of well known "security" headers (for example, involving caching) as specified by Spring Security. The assumption in this case is that the downstream services might add these headers, too, but we want the values from the proxy. To not discard these well known security headers when Spring Security is on the classpath, you can set `zuul.ignoreSecurityHeaders` to `false`. Doing so can be useful if you disabled the HTTP Security response headers in Spring Security and want the values provided by downstream services.

137.6. Management Endpoints

By default, if you use `@EnableZuulProxy` with the Spring Boot Actuator, you enable two additional endpoints:

- Routes
- Filters

137.6.1. Routes Endpoint

A GET to the routes endpoint at `/routes` returns a list of the mapped routes:

GET /routes

```
{
  /stores/**: "http://localhost:8081"
}
```

Additional route details can be requested by adding the `?format=details` query string to `/routes`. Doing so produces the following output:

GET /routes/details

```
{
  "/stores/**": {
    "id": "stores",
    "fullPath": "/stores/**",
    "location": "http://localhost:8081",
    "path": "/**",
    "prefix": "/stores",
    "retryable": false,
    "customSensitiveHeaders": false,
    "prefixStripped": true
  }
}
```

A `POST` to `/routes` forces a refresh of the existing routes (for example, when there have been changes in the service catalog). You can disable this endpoint by setting `endpoints.routes.enabled` to `false`.

the routes should respond automatically to changes in the service catalog, but the `POST` to `/routes` is a way to force the change to happen immediately.

137.6.2. Filters Endpoint

A `GET` to the filters endpoint at `/filters` returns a map of Zuul filters by type. For each filter type in the map, you get a list of all the filters of that type, along with their details.

137.7. Strangulation Patterns and Local Forwards

A common pattern when migrating an existing application or API is to “strangle” old endpoints, slowly replacing them with different implementations. The Zuul proxy is a useful tool for this because you can use it to handle all traffic from the clients of the old endpoints but redirect some of the requests to new ones.

The following example shows the configuration details for a “strangle” scenario:

application.yml

```
zuul:
 routes:
   first:
     path: /first/**
     url: https://first.example.com
   second:
     path: /second/**
     url: forward:/second
   third:
     path: /third/**
     url: forward:/3rd
   legacy:
     path: /**
     url: https://legacy.example.com
```

In the preceding example, we are strangle the “legacy” application, which is mapped to all requests that do not match one of the other patterns. Paths in `/first/**` have been extracted into a new service with an external URL. Paths in `/second/**` are forwarded so that they can be handled locally (for example, with a normal Spring `@RequestMapping`). Paths in `/third/**` are also forwarded but with a different prefix (`/third/foo` is forwarded to `/3rd/foo`).

The ignored patterns aren’t completely ignored, they just are not handled by the proxy (so they are also effectively forwarded locally).

137.8. Uploading Files through Zuul

If you use `@EnableZuulProxy`, you can use the proxy paths to upload files and it should work, so long as the files are small. For large files there is an alternative path that bypasses the Spring `DispatcherServlet` (to avoid multipart processing) in "/zuul/*". In other words, if you have `zuul.routes.customers=/customers/**`, then you can `POST` large files to `/zuul/customers/*`. The servlet path is externalized via `zuul.servletPath`. If the proxy route takes you through a Ribbon load balancer, extremely large files also require elevated timeout settings, as shown in the following example:

application.yml

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000
```

Note that, for streaming to work with large files, you need to use chunked encoding in the request (which some browsers do not do by default), as shown in the following example:

```
$ curl -v -H "Transfer-Encoding: chunked" \
    -F "file=@mylarge.iso" localhost:9999/zuul/simple/file
```

137.9. Query String Encoding

When processing the incoming request, query params are decoded so that they can be available for possible modifications in Zuul filters. They are then re-encoded the back end request is rebuilt in the route filters. The result can be different than the original input if (for example) it was encoded with Javascript's `encodeURIComponent()` method. While this causes no issues in most cases, some web servers can be picky with the encoding of complex query string.

To force the original encoding of the query string, it is possible to pass a special flag to `ZuulProperties` so that the query string is taken as is with the `HttpServletRequest::getQueryString` method, as shown in the following example:

application.yml

```
zuul:
 forceOriginalQueryStringEncoding: true
```

This special flag works only with `SimpleHostRoutingFilter`. Also, you loose the ability to easily override query parameters with `RequestContext.getCurrentContext().setRequestQueryParams(someOverriddenParameters)`, because the query string is now fetched directly on the original `HttpServletRequest`.

137.10. Request URI Encoding

When processing the incoming request, request URI is decoded before matching them to routes. The request URI is then re-encoded when the back end request is rebuilt in the route filters. This can cause some unexpected behavior if your URI includes the encoded "/" character.

To use the original request URI, it is possible to pass a special flag to 'ZuulProperties' so that the URI will be taken as is with the `HttpServletRequest::getRequestURI` method, as shown in the following example:

application.yml

```
zuul:
 decodeUrl: false
```

If you are overriding request URI using `requestURI` RequestContext attribute and this flag is set to false, then the URL set in the request context will not be encoded. It will be your responsibility to make sure the URL is already encoded.

137.11. Plain Embedded Zuul

If you use `@EnableZuulServer` (instead of `@EnableZuulProxy`), you can also run a Zuul server without proxying or selectively switch on parts of the proxying platform. Any beans that you add to the application of type `ZuulFilter` are installed automatically (as they are with `@EnableZuulProxy`) but without any of the proxy filters being added automatically.

In that case, the routes into the Zuul server are still specified by configuring "zuul.routes.*", but there is no service discovery and no proxying. Consequently, the "serviceId" and "url" settings are ignored. The following example maps all paths in "/api/**" to the Zuul filter chain:

application.yml

```
zuul:
 routes:
   api: /api/**
```

137.12. Disable Zuul Filters

Zuul for Spring Cloud comes with a number of `ZuulFilter` beans enabled by default in both proxy and server mode. See the Zuul filters package for the list of filters that you can enable. If you want to disable one, set zuul.<SimpleClassName>.<filterType>.disable=true. By convention, the package after `filters` is the Zuul filter type. For example to disable `org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter`, set `zuul.SendResponseFilter.post.disable=true`.

137.13. Providing Hystrix Fallbacks For Routes

When a circuit for a given route in Zuul is tripped, you can provide a fallback response by creating a bean of type `FallbackProvider`. Within this bean, you need to specify the route ID the fallback is for and provide a `ClientHttpResponse` to return as a fallback. The following example shows a relatively simple `FallbackProvider` implementation:

```
class MyFallbackProvider implements FallbackProvider {

    @Override
    public String getRoute() {
        return "customers";
    }

    @Override
    public ClientHttpResponse fallbackResponse(String route, final Throwable cause) {
        if (cause instanceof HystrixTimeoutException) {
            return response(HttpStatus.GATEWAY_TIMEOUT);
        } else {
            return response(HttpStatus.INTERNAL_SERVER_ERROR);
        }
```

```java
    }

    private ClientHttpResponse response(final HttpStatus status) {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return status;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                return status.value();
            }

            @Override
            public String getStatusText() throws IOException {
                return status.getReasonPhrase();
            }

            @Override
            public void close() {
            }

            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

The following example shows how the route configuration for the previous example might appear:

```yaml
zuul:
  routes:
    customers: /customers/**
```

If you would like to provide a default fallback for all routes, you can create a bean of type `FallbackProvider` and have the `getRoute` method return `*` or `null`, as shown in the following example:

```java
class MyFallbackProvider implements FallbackProvider {
    @Override
    public String getRoute() {
        return "*";
    }

    @Override
    public ClientHttpResponse fallbackResponse(String route, Throwable throwable) {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                return 200;
            }

            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }

            @Override
            public void close() {

            }

            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

137.14. Zuul Timeouts

If you want to configure the socket timeouts and read timeouts for requests proxied through Zuul, you have two options, based on your configuration:

- If Zuul uses service discovery, you need to configure these timeouts with the `ribbon.ReadTimeout` and `ribbon.SocketTimeout` Ribbon properties.

If you have configured Zuul routes by specifying URLs, you need to use `zuul.host.connect-timeout-millis` and `zuul.host.socket-timeout-millis`.

137.15. Rewriting the `Location` header

If Zuul is fronting a web application, you may need to re-write the `Location` header when the web application redirects through a HTTP status code of `3XX`. Otherwise, the browser redirects to the web application's URL instead of the Zuul URL. You can configure a `LocationRewriteFilter` Zuul filter to re-write the `Location` header to the Zuul's URL. It also adds back the stripped global and route-specific prefixes. The following example adds a filter by using a Spring Configuration file:

```
import org.springframework.cloud.netflix.zuul.filters.post.LocationRewriteFilter;
...

@Configuration
@EnableZuulProxy
public class ZuulConfig {
    @Bean
    public LocationRewriteFilter locationRewriteFilter() {
        return new LocationRewriteFilter();
    }
}
```

Use this filter carefully. The filter acts on the `Location` header of ALL `3XX` response codes, which may not be appropriate in all scenarios, such as when redirecting the user to an external URL.

137.16. Enabling Cross Origin Requests

By default Zuul routes all Cross Origin requests (CORS) to the services. If you want instead Zuul to handle these requests it can be done by providing custom `WebMvcConfigurer` bean:

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurer() {
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/path-1/**")
                    .allowedOrigins("https://allowed-origin.com")
                    .allowedMethods("GET", "POST");
        }
    };
}
```

In the example above, we allow `GET` and `POST` methods from `allowed-origin.com` to send cross-origin requests to the endpoints starting with `path-1`. You can apply CORS configuration to a specific path pattern or globally for the whole application, using `/**` mapping. You can customize properties: `allowedOrigins`,`allowedMethods`,`allowedHeaders`,`exposedHeaders`,`allowCredentials` and `maxAge` via this configuration.

137.17. Metrics

Zuul will provide metrics under the Actuator metrics endpoint for any failures that might occur when routing requests. These metrics can be viewed by hitting `/actuator/metrics`. The metrics will have a name that has the format `ZUUL::EXCEPTION:errorCause:statusCode`.

137.18. Zuul Developer Guide

For a general overview of how Zuul works, see the Zuul Wiki.

137.18.1. The Zuul Servlet

Zuul is implemented as a Servlet. For the general cases, Zuul is embedded into the Spring Dispatch mechanism. This lets Spring MVC be in control of the routing. In this case, Zuul buffers requests. If there is a need to go through Zuul without buffering requests (for example, for large file uploads), the Servlet is also installed outside of the Spring Dispatcher. By default, the servlet has an address of `/zuul`. This path can be changed with the `zuul.servlet-path` property.

137.18.2. Zuul RequestContext

To pass information between filters, Zuul uses a `RequestContext`. Its data is held in a `ThreadLocal` specific to each request. Information about where to route requests, errors, and the actual `HttpServletRequest` and `HttpServletResponse` are stored there. The `RequestContext` extends `ConcurrentHashMap`, so anything can be stored in the context. `FilterConstants` contains the keys used by the filters installed by Spring Cloud Netflix (more on these later).

137.18.3. `@EnableZuulProxy` vs. `@EnableZuulServer`

Spring Cloud Netflix installs a number of filters, depending on which annotation was used to enable Zuul. `@EnableZuulProxy` is a superset of `@EnableZuulServer`. In other words, `@EnableZuulProxy` contains all the filters installed by `@EnableZuulServer`. The additional filters in the “proxy” enable routing functionality. If you want a “blank” Zuul, you should use `@EnableZuulServer`.

137.18.4. `@EnableZuulServer` Filters

`@EnableZuulServer` creates a `SimpleRouteLocator` that loads route definitions from Spring Boot configuration files.

The following filters are installed (as normal Spring Beans):

- Pre filters:
  - `ServletDetectionFilter`: Detects whether the request is through the Spring Dispatcher. Sets a

boolean with a key of `FilterConstants.IS_DISPATCHER_SERVLET_REQUEST_KEY`.

  - `FormBodyWrapperFilter`: Parses form data and re-encodes it for downstream requests.
  - `DebugFilter`: If the `debug` request parameter is set, sets `RequestContext.setDebugRouting()` and `RequestContext.setDebugRequest()` to `true`.

- Route filters:
  - `SendForwardFilter`: Forwards requests by using the Servlet `RequestDispatcher`. The forwarding location is stored in the `RequestContext` attribute, `FilterConstants.FORWARD_TO_KEY`. This is useful for forwarding to endpoints in the current application.
- Post filters:
  - `SendResponseFilter`: Writes responses from proxied requests to the current response.
- Error filters:
  - `SendErrorFilter`: Forwards to `/error` (by default) if `RequestContext.getThrowable()` is not null. You can change the default forwarding path (`/error`) by setting the `error.path` property.

137.18.5. `@EnableZuulProxy` Filters

Creates a `DiscoveryClientRouteLocator` that loads route definitions from a `DiscoveryClient` (such as Eureka) as well as from properties. A route is created for each `serviceId` from the `DiscoveryClient`. As new services are added, the routes are refreshed.

In addition to the filters described earlier, the following filters are installed (as normal Spring Beans):

- Pre filters:
  - `PreDecorationFilter`: Determines where and how to route, depending on the supplied `RouteLocator`. It also sets various proxy-related headers for downstream requests.
- Route filters:
  - `RibbonRoutingFilter`: Uses Ribbon, Hystrix, and pluggable HTTP clients to send requests. Service IDs are found in the `RequestContext` attribute, `FilterConstants.SERVICE_ID_KEY`. This filter can use different HTTP clients:
    - Apache `HttpClient`: The default client.
    - Squareup `OkHttpClient` v3: Enabled by having the `com.squareup.okhttp3:okhttp` library on the classpath and setting `ribbon.okhttp.enabled=true`.
    - Netflix Ribbon HTTP client: Enabled by setting `ribbon.restclient.enabled=true`. This client has limitations, including that it does not support the PATCH method, but it also has built-in retry.
  - `SimpleHostRoutingFilter`: Sends requests to predetermined URLs through an Apache HttpClient. URLs are found in `RequestContext.getRouteHost()`.

137.18.6. Custom Zuul Filter Examples

Most of the following "How to Write" examples below are included Sample Zuul Filters project. There are also examples of manipulating the request or response body in that repository.

This section includes the following examples:

- How to Write a Pre Filter
- How to Write a Route Filter
- How to Write a Post Filter

How to Write a Pre Filter

Pre filters set up data in the `RequestContext` for use in filters downstream. The main use case is to set information required for route filters. The following example shows a Zuul pre filter:

```
public class QueryParamPreFilter extends ZuulFilter {
    @Override
    public int filterOrder() {
        return PRE_DECORATION_FILTER_ORDER - 1; // run before PreDecoration
    }

    @Override
    public String filterType() {
        return PRE_TYPE;
    }

    @Override
    public boolean shouldFilter() {
        RequestContext ctx = RequestContext.getCurrentContext();
        return !ctx.containsKey(FORWARD_TO_KEY) // a filter has already forwarded
                && !ctx.containsKey(SERVICE_ID_KEY); // a filter has already
determined serviceId
    }
    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        if (request.getParameter("sample") != null) {
            // put the serviceId in `RequestContext`
            ctx.put(SERVICE_ID_KEY, request.getParameter("foo"));
        }
        return null;
    }
}
```

The preceding filter populates `SERVICE_ID_KEY` from the `sample` request parameter. In practice, you should not do that kind of direct mapping. Instead, the service ID should be looked up from the value of `sample` instead.

Now that `SERVICE_ID_KEY` is populated, `PreDecorationFilter` does not run and `RibbonRoutingFilter` runs.

If you want to route to a full URL, call `ctx.setRouteHost(url)` instead.

To modify the path to which routing filters forward, set the `REQUEST_URI_KEY`.

How to Write a Route Filter

Route filters run after pre filters and make requests to other services. Much of the work here is to translate request and response data to and from the model required by the client. The following example shows a Zuul route filter:

```
public class OkHttpRoutingFilter extends ZuulFilter {
    @Autowired
    private ProxyRequestHelper helper;

    @Override
    public String filterType() {
        return ROUTE_TYPE;
    }

    @Override
    public int filterOrder() {
        return SIMPLE_HOST_ROUTING_FILTER_ORDER - 1;
    }

    @Override
    public boolean shouldFilter() {
        return RequestContext.getCurrentContext().getRouteHost() != null
                && RequestContext.getCurrentContext().sendZuulResponse();
    }

    @Override
    public Object run() {
        OkHttpClient httpClient = new OkHttpClient.Builder()
                // customize
                .build();

        RequestContext context = RequestContext.getCurrentContext();
        HttpServletRequest request = context.getRequest();

        String method = request.getMethod();

        String uri = this.helper.buildZuulRequestURI(request);

        Headers.Builder headers = new Headers.Builder();
        Enumeration<String> headerNames = request.getHeaderNames();
        while (headerNames.hasMoreElements()) {
            String name = headerNames.nextElement();
```

```
            Enumeration<String> values = request.getHeaders(name);

            while (values.hasMoreElements()) {
                String value = values.nextElement();
                headers.add(name, value);
            }
        }

        InputStream inputStream = request.getInputStream();

        RequestBody requestBody = null;
        if (inputStream != null && HttpMethod.permitsRequestBody(method)) {
            MediaType mediaType = null;
            if (headers.get("Content-Type") != null) {
                mediaType = MediaType.parse(headers.get("Content-Type"));
            }
            requestBody = RequestBody.create(mediaType,
StreamUtils.copyToByteArray(inputStream));
        }

        Request.Builder builder = new Request.Builder()
                .headers(headers.build())
                .url(uri)
                .method(method, requestBody);

        Response response = httpClient.newCall(builder.build()).execute();

        LinkedMultiValueMap<String, String> responseHeaders = new
LinkedMultiValueMap<>();

        for (Map.Entry<String, List<String>> entry :
response.headers().toMultimap().entrySet()) {
            responseHeaders.put(entry.getKey(), entry.getValue());
        }

        this.helper.setResponse(response.code(), response.body().byteStream(),
                responseHeaders);
        context.setRouteHost(null); // prevent SimpleHostRoutingFilter from running
        return null;
    }
}
```

The preceding filter translates Servlet request information into OkHttp3 request information, executes an HTTP request, and translates OkHttp3 response information to the Servlet response.

How to Write a Post Filter

Post filters typically manipulate the response. The following filter adds a random `UUID` as the `X-Sample` header:

```
public class AddResponseHeaderFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return POST_TYPE;
    }

    @Override
    public int filterOrder() {
        return SEND_RESPONSE_FILTER_ORDER - 1;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext context = RequestContext.getCurrentContext();
        HttpServletResponse servletResponse = context.getResponse();
        servletResponse.addHeader("X-Sample", UUID.randomUUID().toString());
        return null;
    }
}
```

Other manipulations, such as transforming the response body, are much more complex and computationally intensive.

137.18.7. How Zuul Errors Work

If an exception is thrown during any portion of the Zuul filter lifecycle, the error filters are executed. The `SendErrorFilter` is only run if `RequestContext.getThrowable()` is not `null`. It then sets specific `javax.servlet.error.*` attributes in the request and forwards the request to the Spring Boot error page.

137.18.8. Zuul Eager Application Context Loading

Zuul internally uses Ribbon for calling the remote URLs. By default, Ribbon clients are lazily loaded by Spring Cloud on first call. This behavior can be changed for Zuul by using the following configuration, which results eager loading of the child Ribbon related Application contexts at application startup time. The following example shows how to enable eager loading:

application.yml

```
zuul:
  ribbon:
    eager-load:
      enabled: true
```

Chapter 138. Polyglot support with Sidecar

Do you have non-JVM languages with which you want to take advantage of Eureka, Ribbon, and Config Server? The Spring Cloud Netflix Sidecar was inspired by Netflix Prana. It includes an HTTP API to get all of the instances (by host and port) for a given service. You can also proxy service calls through an embedded Zuul proxy that gets its route entries from Eureka. The Spring Cloud Config Server can be accessed directly through host lookup or through the Zuul Proxy. The non-JVM application should implement a health check so the Sidecar can report to Eureka whether the app is up or down.

To include Sidecar in your project, use the dependency with a group ID of `org.springframework.cloud` and artifact ID or `spring-cloud-netflix-sidecar`.

To enable the Sidecar, create a Spring Boot application with `@EnableSidecar`. This annotation includes `@EnableCircuitBreaker`, `@EnableDiscoveryClient`, and `@EnableZuulProxy`. Run the resulting application on the same host as the non-JVM application.

To configure the side car, add `sidecar.port` and `sidecar.health-uri` to `application.yml`. The `sidecar.port` property is the port on which the non-JVM application listens. This is so the Sidecar can properly register the application with Eureka. The `sidecar.secure-port-enabled` options provides a way to enable secure port for traffic. The `sidecar.health-uri` is a URI accessible on the non-JVM application that mimics a Spring Boot health indicator. It should return a JSON document that resembles the following:

health-uri-document

```
{
  "status":"UP"
}
```

The following application.yml example shows sample configuration for a Sidecar application:

application.yml

```
server:
  port: 5678
spring:
  application:
    name: sidecar

sidecar:
  port: 8000
  health-uri: http://localhost:8000/health.json
```

The API for the `DiscoveryClient.getInstances()` method is `/hosts/{serviceId}`. The following example response for `/hosts/customers` returns two instances on different hosts:

/hosts/customers

```
[
    {
        "host": "myhost",
        "port": 9000,
        "uri": "https://myhost:9000",
        "serviceId": "CUSTOMERS",
        "secure": false
    },
    {
        "host": "myhost2",
        "port": 9000,
        "uri": "https://myhost2:9000",
        "serviceId": "CUSTOMERS",
        "secure": false
    }
]
```

This API is accessible to the non-JVM application (if the sidecar is on port 5678) at `localhost:5678/hosts/{serviceId}`.

The Zuul proxy automatically adds routes for each service known in Eureka to `/<serviceId>`, so the customers service is available at `/customers`. The non-JVM application can access the customer service at `localhost:5678/customers` (assuming the sidecar is listening on port 5678).

If the Config Server is registered with Eureka, the non-JVM application can access it through the Zuul proxy. If the `serviceId` of the ConfigServer is `configserver` and the Sidecar is on port 5678, then it can be accessed at localhost:5678/configserver.

Non-JVM applications can take advantage of the Config Server's ability to return YAML documents. For example, a call to sidecar.local.spring.io:5678/configserver/default-master.yml might result in a YAML document resembling the following:

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  password: password
info:
  description: Spring Cloud Samples
  url: https://github.com/spring-cloud-samples
```

To enable the health check request to accept all certificates when using HTTPs set `sidecar.accept-all-ssl-certificates` to `true.

Chapter 139. Retrying Failed Requests

Spring Cloud Netflix offers a variety of ways to make HTTP requests. You can use a load balanced `RestTemplate`, Ribbon, or Feign. No matter how you choose to create your HTTP requests, there is always a chance that a request may fail. When a request fails, you may want to have the request be retried automatically. To do so when using Sping Cloud Netflix, you need to include Spring Retry on your application's classpath. When Spring Retry is present, load-balanced `RestTemplates`, Feign, and Zuul automatically retry any failed requests (assuming your configuration allows doing so).

139.1. BackOff Policies

By default, no backoff policy is used when retrying requests. If you would like to configure a backoff policy, you need to create a bean of type `LoadBalancedRetryFactory` and override the `createBackOffPolicy` method for a given service, as shown in the following example:

```
@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedRetryFactory retryFactory() {
        return new LoadBalancedRetryFactory() {
            @Override
            public BackOffPolicy createBackOffPolicy(String service) {
                return new ExponentialBackOffPolicy();
            }
        };
    }
}
```

139.2. Configuration

When you use Ribbon with Spring Retry, you can control the retry functionality by configuring certain Ribbon properties. To do so, set the `client.ribbon.MaxAutoRetries`, `client.ribbon.MaxAutoRetriesNextServer`, and `client.ribbon.OkToRetryOnAllOperations` properties. See the Ribbon documentation for a description of what these properties do.

Enabling `client.ribbon.OkToRetryOnAllOperations` includes retrying POST requests, which can have an impact on the server's resources, due to the buffering of the request body.

The property names are case-sensitive, and since some of these properties are defined in the Netflix Ribbon project, they are in Pascal Case and the ones from Spring Cloud are in Camel Case.

In addition, you may want to retry requests when certain status codes are returned in the response. You can list the response codes you would like the Ribbon client to retry by setting the `clientName.ribbon.retryableStatusCodes` property, as shown in the following example:

```
clientName:
  ribbon:
    retryableStatusCodes: 404,502
```

You can also create a bean of type `LoadBalancedRetryPolicy` and implement the `retryableStatusCode` method to retry a request given the status code.

139.2.1. Zuul

You can turn off Zuul's retry functionality by setting `zuul.retryable` to `false`. You can also disable retry functionality on a route-by-route basis by setting `zuul.routes.routename.retryable` to `false`.

Chapter 140. HTTP Clients

Spring Cloud Netflix automatically creates the HTTP client used by Ribbon, Feign, and Zuul for you. However, you can also provide your own HTTP clients customized as you need them to be. To do so, you can create a bean of type `CloseableHttpClient` if you are using the Apache HTTP Client or `OkHttpClient` if you are using OK HTTP.

When you create your own HTTP client, you are also responsible for implementing the correct connection management strategies for these clients. Doing so improperly can result in resource management issues.

Chapter 141. Modules In Maintenance Mode

Placing a module in maintenance mode means that the Spring Cloud team will no longer be adding new features to the module. We will fix blocker bugs and security issues, and we will also consider and review small pull requests from the community.

We intend to continue to support these modules for a period of at least a year from the general availability of the Greenwich release train.

The following Spring Cloud Netflix modules and corresponding starters will be placed into maintenance mode:

- spring-cloud-netflix-archaius
- spring-cloud-netflix-concurrency-limits
- spring-cloud-netflix-hystrix-contract
- spring-cloud-netflix-hystrix-dashboard
- spring-cloud-netflix-hystrix-stream
- spring-cloud-netflix-hystrix
- spring-cloud-netflix-ribbon
- spring-cloud-netflix-turbine-stream
- spring-cloud-netflix-turbine
- spring-cloud-netflix-zuul

This does not include the Eureka modules.

Chapter 142. Configuration properties

To see the list of all Spring Cloud Netflix related configuration properties please check the Appendix page.

Spring Cloud OpenFeign

Hoxton.SR8

This project provides OpenFeign integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

Chapter 143. Declarative REST Client: Feign

Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same `HttpMessageConverters` used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka, as well as Spring Cloud LoadBalancer to provide a load-balanced http client when using Feign.

143.1. How to Include Feign

To include Feign in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-openfeign`. See the Spring Cloud Project page for details on setting up your build system with the current Spring Cloud Release Train.

Example spring boot app

```
@SpringBootApplication
@EnableFeignClients
public class Application {

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

}
```

StoreClient.java

```
@FeignClient("stores")
public interface StoreClient {
    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    List<Store> getStores();

    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    Page<Store> getStores(Pageable pageable);

    @RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes
= "application/json")
    Store update(@PathVariable("storeId") Long storeId, Store store);
}
```

In the `@FeignClient` annotation the String value ("stores" above) is an arbitrary client name, which is used to create either a Ribbon load-balancer (see below for details of Ribbon support) or Spring Cloud LoadBalancer. You can also specify a URL using the `url` attribute (absolute value or just a hostname). The name of the bean in the application context is the fully qualified name of the interface. To specify your own alias value you can use the `qualifier` value of the `@FeignClient`

annotation.

The load-balancer client above will want to discover the physical addresses for the "stores" service. If your application is a Eureka client then it will resolve the service in the Eureka service registry. If you don't want to use Eureka, you can simply configure a list of servers in your external configuration using `SimpleDiscoveryClient`.

In order to maintain backward compatibility, is used as the default load-balancer implementation. However, Spring Cloud Netflix Ribbon is now in maintenance mode, so we recommend using Spring Cloud LoadBalancer instead. To do this, set the value of `spring.cloud.loadbalancer.ribbon.enabled` to `false`.

143.2. Overriding Feign Defaults

A central concept in Spring Cloud's Feign support is that of the named client. Each feign client is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer using the `@FeignClient` annotation. Spring Cloud creates a new ensemble as an `ApplicationContext` on demand for each named client using `FeignClientsConfiguration`. This contains (amongst other things) an `feign.Decoder`, a `feign.Encoder`, and a `feign.Contract`. It is possible to override the name of that ensemble by using the `contextId` attribute of the `@FeignClient` annotation.

Spring Cloud lets you take full control of the feign client by declaring additional configuration (on top of the `FeignClientsConfiguration`) using `@FeignClient`. Example:

```
@FeignClient(name = "stores", configuration = FooConfiguration.class)
public interface StoreClient {
    //..
}
```

In this case the client is composed from the components already in `FeignClientsConfiguration` together with any in `FooConfiguration` (where the latter will override the former).

`FooConfiguration` does not need to be annotated with `@Configuration`. However, if it is, then take care to exclude it from any `@ComponentScan` that would otherwise include this configuration as it will become the default source for `feign.Decoder`, `feign.Encoder`, `feign.Contract`, etc., when specified. This can be avoided by putting it in a separate, non-overlapping package from any `@ComponentScan` or `@SpringBootApplication`, or it can be explicitly excluded in `@ComponentScan`.

The `serviceId` attribute is now deprecated in favor of the `name` attribute.

Using `contextId` attribute of the `@FeignClient` annotation in addition to changing the name of the `ApplicationContext` ensemble, it will override the alias of the client name and it will be used as part of the name of the configuration bean created for that client.

Previously, using the `url` attribute, did not require the `name` attribute. Using `name` is now required.

Placeholders are supported in the `name` and `url` attributes.

```
@FeignClient(name = "${feign.name}", url = "${feign.url}")
public interface StoreClient {
    //..
}
```

Spring Cloud OpenFeign provides the following beans by default for feign (`BeanType` beanName: `ClassName`):

- `Decoder` feignDecoder: `ResponseEntityDecoder` (which wraps a `SpringDecoder`)
- `Encoder` feignEncoder: `SpringEncoder`
- `Logger` feignLogger: `Slf4jLogger`
- `Contract` feignContract: `SpringMvcContract`
- `Feign.Builder` feignBuilder: `HystrixFeign.Builder`
- `Client` feignClient: if Ribbon is in the classpath and is enabled it is a `LoadBalancerFeignClient`, otherwise if Spring Cloud LoadBalancer is in the classpath, `FeignBlockingLoadBalancerClient` is used. If none of them is in the classpath, the default feign client is used.

`spring-cloud-starter-openfeign` supports both `spring-cloud-starter-netflix-ribbon` and `spring-cloud-starter-loadbalancer`. However, as they are optional dependencies, you need to make sure the one you want to use has been added to your project.

The OkHttpClient and ApacheHttpClient feign clients can be used by setting `feign.okhttp.enabled` or `feign.httpclient.enabled` to `true`, respectively, and having them on the classpath. You can customize the HTTP client used by providing a bean of either `org.apache.http.impl.client.CloseableHttpClient` when using Apache or `okhttp3.OkHttpClient` when using OK HTTP.

Spring Cloud OpenFeign *does not* provide the following beans by default for feign, but still looks up beans of these types from the application context to create the feign client:

- `Logger.Level`
- `Retryer`
- `ErrorDecoder`
- `Request.Options`
- `Collection<RequestInterceptor>`
- `SetterFactory`
- `QueryMapEncoder`

A bean of `Retryer.NEVER_RETRY` with the type `Retryer` is created by default, which will disable

retrying. Notice this retrying behavior is different from the Feign default one, where it will automatically retry IOExceptions, treating them as transient network related exceptions, and any RetryableException thrown from an ErrorDecoder.

Creating a bean of one of those type and placing it in a `@FeignClient` configuration (such as `FooConfiguration` above) allows you to override each one of the beans described. Example:

```
@Configuration
public class FooConfiguration {
    @Bean
    public Contract feignContract() {
        return new feign.Contract.Default();
    }

    @Bean
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
        return new BasicAuthRequestInterceptor("user", "password");
    }
}
```

This replaces the `SpringMvcContract` with `feign.Contract.Default` and adds a `RequestInterceptor` to the collection of `RequestInterceptor`.

`@FeignClient` also can be configured using configuration properties.

application.yml

```
feign:
  client:
    config:
      feignName:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: full
        errorDecoder: com.example.SimpleErrorDecoder
        retryer: com.example.SimpleRetryer
        requestInterceptors:
          - com.example.FooRequestInterceptor
          - com.example.BarRequestInterceptor
        decode404: false
        encoder: com.example.SimpleEncoder
        decoder: com.example.SimpleDecoder
        contract: com.example.SimpleContract
```

Default configurations can be specified in the `@EnableFeignClients` attribute `defaultConfiguration` in a similar manner as described above. The difference is that this configuration will apply to *all* feign clients.

If you prefer using configuration properties to configured all `@FeignClient`, you can create

configuration properties with `default` feign name.

application.yml

```
feign:
  client:
    config:
      default:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: basic
```

If we create both `@Configuration` bean and configuration properties, configuration properties will win. It will override `@Configuration` values. But if you want to change the priority to `@Configuration`, you can change `feign.client.default-to-properties` to `false`.

If you need to use `ThreadLocal` bound variables in your `RequestInterceptor`s you will need to either set the thread isolation strategy for Hystrix to `SEMAPHORE` or disable Hystrix in Feign.

application.yml

```
# To disable Hystrix in Feign
feign:
  hystrix:
    enabled: false

# To set thread isolation to SEMAPHORE
hystrix:
  command:
    default:
      execution:
        isolation:
          strategy: SEMAPHORE
```

If we want to create multiple feign clients with the same name or url so that they would point to the same server but each with a different custom configuration then we have to use `contextId` attribute of the `@FeignClient` in order to avoid name collision of these configuration beans.

```
@FeignClient(contextId = "fooClient", name = "stores", configuration =
FooConfiguration.class)
public interface FooClient {
    //..
}
```

```
@FeignClient(contextId = "barClient", name = "stores", configuration =
BarConfiguration.class)
public interface BarClient {
    //..
}
```

It is also possible to configure FeignClient not to inherit beans from the parent context. You can do this by overriding the `inheritParentConfiguration()` in a `FeignClientConfigurer` bean to return `false`:

```
@Configuration
public class CustomConfiguration{

@Bean
public FeignClientConfigurer feignClientConfigurer() {
            return new FeignClientConfigurer() {

                @Override
                public boolean inheritParentConfiguration() {
                    return false;
                }
            };

        }
}
```

143.3. Creating Feign Clients Manually

In some cases it might be necessary to customize your Feign Clients in a way that is not possible using the methods above. In this case you can create Clients using the Feign Builder API. Below is an example which creates two Feign Clients with the same interface but configures each one with a separate request interceptor.

```
@Import(FeignClientsConfiguration.class)
class FooController {

    private FooClient fooClient;

    private FooClient adminClient;

        @Autowired
    public FooController(Decoder decoder, Encoder encoder, Client client, Contract
contract) {
        this.fooClient = Feign.builder().client(client)
                .encoder(encoder)
                .decoder(decoder)
                .contract(contract)
                .requestInterceptor(new BasicAuthRequestInterceptor("user", "user"))
                .target(FooClient.class, "https://PROD-SVC");

        this.adminClient = Feign.builder().client(client)
                .encoder(encoder)
                .decoder(decoder)
                .contract(contract)
                .requestInterceptor(new BasicAuthRequestInterceptor("admin", "admin"))
                .target(FooClient.class, "https://PROD-SVC");
    }
}
```

In the above example `FeignClientsConfiguration.class` is the default configuration provided by Spring Cloud OpenFeign.

`PROD-SVC` is the name of the service the Clients will be making requests to.

The Feign `Contract` object defines what annotations and values are valid on interfaces. The autowired `Contract` bean provides supports for SpringMVC annotations, instead of the default Feign native annotations.

You can also use the `Builder`to configure FeignClient not to inherit beans from the parent context. You can do this by overriding calling `inheritParentContext(false)` on the `Builder`.

143.4. Feign Hystrix Support

If Hystrix is on the classpath and `feign.hystrix.enabled=true`, Feign will wrap all methods with a circuit breaker. Returning a `com.netflix.hystrix.HystrixCommand` is also available. This lets you use reactive patterns (with a call to `.toObservable()` or `.observe()` or asynchronous use (with a call to `.queue()`).

To disable Hystrix support on a per-client basis create a vanilla `Feign.Builder` with the "prototype" scope, e.g.:

```
@Configuration
public class FooConfiguration {
        @Bean
    @Scope("prototype")
    public Feign.Builder feignBuilder() {
        return Feign.builder();
    }
}
```

Prior to the Spring Cloud Dalston release, if Hystrix was on the classpath Feign would have wrapped all methods in a circuit breaker by default. This default behavior was changed in Spring Cloud Dalston in favor for an opt-in approach.

143.5. Feign Hystrix Fallbacks

Hystrix supports the notion of a fallback: a default code path that is executed when they circuit is open or there is an error. To enable fallbacks for a given `@FeignClient` set the `fallback` attribute to the class name that implements the fallback. You also need to declare your implementation as a Spring bean.

```
@FeignClient(name = "hello", fallback = HystrixClientFallback.class)
protected interface HystrixClient {
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    Hello iFailSometimes();
}

static class HystrixClientFallback implements HystrixClient {
    @Override
    public Hello iFailSometimes() {
        return new Hello("fallback");
    }
}
```

If one needs access to the cause that made the fallback trigger, one can use the `fallbackFactory` attribute inside `@FeignClient`.

```
@FeignClient(name = "hello", fallbackFactory = HystrixClientFallbackFactory.class)
protected interface HystrixClient {
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    Hello iFailSometimes();
}

@Component
static class HystrixClientFallbackFactory implements FallbackFactory<HystrixClient> {
    @Override
    public HystrixClient create(Throwable cause) {
        return new HystrixClient() {
            @Override
            public Hello iFailSometimes() {
                return new Hello("fallback; reason was: " + cause.getMessage());
            }
        };
    }
}
```

There is a limitation with the implementation of fallbacks in Feign and how Hystrix fallbacks work. Fallbacks are currently not supported for methods that return `com.netflix.hystrix.HystrixCommand` and `rx.Observable`.

143.6. Feign and `@Primary`

When using Feign with Hystrix fallbacks, there are multiple beans in the `ApplicationContext` of the same type. This will cause `@Autowired` to not work because there isn't exactly one bean, or one marked as primary. To work around this, Spring Cloud OpenFeign marks all Feign instances as `@Primary`, so Spring Framework will know which bean to inject. In some cases, this may not be desirable. To turn off this behavior set the `primary` attribute of `@FeignClient` to false.

```
@FeignClient(name = "hello", primary = false)
public interface HelloClient {
    // methods here
}
```

143.7. Feign Inheritance Support

Feign supports boilerplate apis via single-inheritance interfaces. This allows grouping common operations into convenient base interfaces.

UserService.java

```
public interface UserService {

    @RequestMapping(method = RequestMethod.GET, value ="/users/{id}")
    User getUser(@PathVariable("id") long id);
}
```

UserResource.java

```
@RestController
public class UserResource implements UserService {

}
```

UserClient.java

```
package project.user;

@FeignClient("users")
public interface UserClient extends UserService {

}
```

It is generally not advisable to share an interface between a server and a client. It introduces tight coupling, and also actually doesn't work with Spring MVC in its current form (method parameter mapping is not inherited).

143.8. Feign request/response compression

You may consider enabling the request or response GZIP compression for your Feign requests. You can do this by enabling one of the properties:

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
```

Feign request compression gives you settings similar to what you may set for your web server:

```
feign.compression.request.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

These properties allow you to be selective about the compressed media types and minimum request threshold length.

For http clients except OkHttpClient, default gzip decoder can be enabled to decode gzip response in UTF-8 encoding:

```
feign.compression.response.enabled=true
feign.compression.response.useGzipDecoder=true
```

143.9. Feign logging

A logger is created for each Feign client created. By default the name of the logger is the full class name of the interface used to create the Feign client. Feign logging only responds to the `DEBUG` level.

application.yml

```
logging.level.project.user.UserClient: DEBUG
```

The `Logger.Level` object that you may configure per client, tells Feign how much to log. Choices are:

- `NONE`, No logging (**DEFAULT**).
- `BASIC`, Log only the request method and URL and the response status code and execution time.
- `HEADERS`, Log the basic information along with request and response headers.
- `FULL`, Log the headers, body, and metadata for both requests and responses.

For example, the following would set the `Logger.Level` to `FULL`:

```
@Configuration
public class FooConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```

143.10. Feign @QueryMap support

The OpenFeign `@QueryMap` annotation provides support for POJOs to be used as GET parameter maps. Unfortunately, the default OpenFeign QueryMap annotation is incompatible with Spring because it lacks a `value` property.

Spring Cloud OpenFeign provides an equivalent `@SpringQueryMap` annotation, which is used to annotate a POJO or Map parameter as a query parameter map.

For example, the `Params` class defines parameters `param1` and `param2`:

```
// Params.java
public class Params {
    private String param1;
    private String param2;

    // [Getters and setters omitted for brevity]
}
```

The following feign client uses the `Params` class by using the `@SpringQueryMap` annotation:

```
@FeignClient("demo")
public interface DemoTemplate {

    @GetMapping(path = "/demo")
    String demoEndpoint(@SpringQueryMap Params params);
}
```

If you need more control over the generated query parameter map, you can implement a custom `QueryMapEncoder` bean.

143.11. HATEOAS support

Spring provides some APIs to create REST representations that follow the HATEOAS principle, Spring Hateoas and Spring Data REST.

If your project use the `org.springframework.boot:spring-boot-starter-hateoas` starter or the `org.springframework.boot:spring-boot-starter-data-rest` starter, Feign HATEOAS support is enabled by default.

When HATEOAS support is enabled, Feign clients are allowed to serialize and deserialize HATEOAS representation models: EntityModel, CollectionModel and PagedModel.

```
@FeignClient("demo")
public interface DemoTemplate {

    @GetMapping(path = "/stores")
    CollectionModel<Store> getStores();
}
```

143.12. Spring @MatrixVariable Support

Spring Cloud OpenFeign provides support for the Spring `@MatrixVariable` annotation.

If a map is passed as the method argument, the `@MatrixVariable` path segment is created by joining key-value pairs from the map with a `=`.

If a different object is passed, either the `name` provided in the `@MatrixVariable` annotation (if defined) or the annotated variable name is joined with the provided method argument using `=`.

IMPORTANT

Even though, on the server side, Spring does not require the users to name the path segment placeholder same as the matrix variable name, since it would be too ambiguous on the client side, Sprig Cloud OpenFeign requires that you add a path segment placeholder with a name matching either the `name` provided in the `@MatrixVariable` annotation (if defined) or the annotated variable name.

For example:

```
@GetMapping("/objects/links/{matrixVars}")
Map<String, List<String>> getObjects(@MatrixVariable Map<String, List<String>>
matrixVars);
```

Note that both variable name and the path segment placeholder are called `matrixVars`.

```
@FeignClient("demo")
public interface DemoTemplate {

    @GetMapping(path = "/stores")
    CollectionModel<Store> getStores();
}
```

143.13. Feign `CollectionFormat` support

We support `feign.CollectionFormat` by providing the `@CollectionFormat` annotation. You can annotate a Feign client method with it by passing the desired `feign.CollectionFormat` as annotation value.

In the following example, the `CSV` format is used instead of the default `EXPLODED` to process the method.

```
@FeignClient(name = "demo")
    protected interface PageableFeignClient {

        @CollectionFormat(feign.CollectionFormat.CSV)
        @GetMapping(path = "/page")
        ResponseEntity performRequest(Pageable page);

    }
```

Set the `CSV` format while sending `Pageable` as a query parameter in order for it to be encoded correctly.

143.14. Reactive Support

As the OpenFeign project does not currently support reactive clients, such as Spring WebClient, neither does Spring Cloud OpenFeign. We will add support for it here as soon as it becomes available in the core project.

Until that is done, we recommend using feign-reactive for Spring WebClient support.

143.14.1. Early Initialization Errors

Depending on how you are using your Feign clients you may see initialization errors when starting your application. To work around this problem you can use an `ObjectProvider` when autowiring your client.

```
@Autowired
ObjectProvider<TestFeginClient> testFeginClient;
```

Chapter 144. Configuration properties

To see the list of all Sleuth related configuration properties please check the Appendix page.

Spring Cloud Security

Spring Cloud Security offers a set of primitives for building secure applications and services with minimum fuss. A declarative model which can be heavily configured externally (or centrally) lends itself to the implementation of large systems of co-operating, remote components, usually with a central indentity management service. It is also extremely easy to use in a service platform like Cloud Foundry. Building on Spring Boot and Spring Security OAuth2 we can quickly create systems that implement common patterns like single sign on, token relay and token exchange.

In a future major release, the functionality contained in this project will move to the respective projects.

Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at github.

Chapter 145. Quickstart

145.1. OAuth2 Single Sign On

Here's a Spring Cloud "Hello World" app with HTTP Basic authentication and a single user account:

app.groovy

```
@Grab('spring-boot-starter-security')
@Controller
class Application {

  @RequestMapping('/')
  String home() {
    'Hello World'
  }

}
```

You can run it with `spring run app.groovy` and watch the logs for the password (username is "user"). So far this is just the default for a Spring Boot app.

Here's a Spring Cloud app with OAuth2 SSO:

app.groovy

```
@Controller
@EnableOAuth2Sso
class Application {

  @RequestMapping('/')
  String home() {
    'Hello World'
  }

}
```

Spot the difference? This app will actually behave exactly the same as the previous one, because it doesn't know it's OAuth2 credentals yet.

You can register an app in github quite easily, so try that if you want a production app on your own domain. If you are happy to test on localhost:8080, then set up these properties in your application configuration:

application.yml

```
security:
  oauth2:
    client:
      clientId: bd1c0a783ccdd1c9b9e4
      clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1
      accessTokenUri: https://github.com/login/oauth/access_token
      userAuthorizationUri: https://github.com/login/oauth/authorize
      clientAuthenticationScheme: form
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

run the app above and it will redirect to github for authorization. If you are already signed into github you won't even notice that it has authenticated. These credentials will only work if your app is running on port 8080.

To limit the scope that the client asks for when it obtains an access token you can set `security.oauth2.client.scope` (comma separated or an array in YAML). By default the scope is empty and it is up to to Authorization Server to decide what the defaults should be, usually depending on the settings in the client registration that it holds.

The examples above are all Groovy scripts. If you want to write the same code in Java (or Groovy) you need to add Spring Security OAuth2 to the classpath (e.g. see the sample here).

145.2. OAuth2 Protected Resource

You want to protect an API resource with an OAuth2 token? Here's a simple example (paired with the client above):

app.groovy

```
@Grab('spring-cloud-starter-security')
@RestController
@EnableResourceServer
class Application {

  @RequestMapping('/')
  def home() {
    [message: 'Hello World']
  }

}
```

and

application.yml

```
security:
  oauth2:
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

Chapter 146. More Detail

146.1. Single Sign On

All of the OAuth2 SSO and resource server features moved to Spring Boot in version 1.3. You can find documentation in the Spring Boot user guide.

146.2. Token Relay

A Token Relay is where an OAuth2 consumer acts as a Client and forwards the incoming token to outgoing resource requests. The consumer can be a pure Client (like an SSO application) or a Resource Server.

146.2.1. Client Token Relay in Spring Cloud Gateway

If your app also has a Spring Cloud Gateway embedded reverse proxy then you can ask it to forward OAuth2 access tokens downstream to the services it is proxying. Thus the SSO app above can be enhanced simply like this:

App.java

```
@Autowired
private TokenRelayGatewayFilterFactory filterFactory;

@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
            .route("resource", r -> r.path("/resource")
                    .filters(f -> f.filter(filterFactory.apply()))
                    .uri("http://localhost:9000"))
            .build();
}
```

or this

application.yaml

```
spring:
  cloud:
    gateway:
      routes:
      - id: resource
        uri: http://localhost:9000
        predicates:
        - Path=/resource
        filters:
        - TokenRelay=
```

and it will (in addition to logging the user in and grabbing a token) pass the authentication token downstream to the services (in this case `/resource`).

To enable this for Spring Cloud Gateway add the following dependencies

- `org.springframework.boot:spring-boot-starter-oauth2-client`
- `org.springframework.cloud:spring-cloud-starter-security`

How does it work? The filter extracts an access token from the currently authenticated user, and puts it in a request header for the downstream requests.

For a full working sample see this project.

The default implementation of `ReactiveOAuth2AuthorizedClientService` used by `TokenRelayGatewayFilterFactory` uses an in-memory data store. You will need to provide your own implementation `ReactiveOAuth2AuthorizedClientService` if you need a more robust solution.

146.2.2. Client Token Relay

If your app is a user facing OAuth2 client (i.e. has declared `@EnableOAuth2Sso` or `@EnableOAuth2Client`) then it has an `OAuth2ClientContext` in request scope from Spring Boot. You can create your own `OAuth2RestTemplate` from this context and an autowired `OAuth2ProtectedResourceDetails`, and then the context will always forward the access token downstream, also refreshing the access token automatically if it expires. (These are features of Spring Security and Spring Boot.)

Spring Boot (1.4.1) does not create an `OAuth2ProtectedResourceDetails` automatically if you are using `client_credentials` tokens. In that case you need to create your own `ClientCredentialsResourceDetails` and configure it with `@ConfigurationProperties("security.oauth2.client")`.

146.2.3. Client Token Relay in Zuul Proxy

If your app also has a Spring Cloud Zuul embedded reverse proxy (using `@EnableZuulProxy`) then you can ask it to forward OAuth2 access tokens downstream to the services it is proxying. Thus the SSO app above can be enhanced simply like this:

app.groovy

```
@Controller
@EnableOAuth2Sso
@EnableZuulProxy
class Application {

}
```

and it will (in addition to logging the user in and grabbing a token) pass the authentication token downstream to the `/proxy/*` services. If those services are implemented with `@EnableResourceServer` then they will get a valid token in the correct header.

How does it work? The `@EnableOAuth2Sso` annotation pulls in `spring-cloud-starter-security` (which you could do manually in a traditional app), and that in turn triggers some autoconfiguration for a `ZuulFilter`, which itself is activated because Zuul is on the classpath (via `@EnableZuulProxy`). The filter just extracts an access token from the currently authenticated user, and puts it in a request header for the downstream requests.

Spring Boot does not create an `OAuth2RestOperations` automatically which is needed for `refresh_token`. In that case you need to create your own `OAuth2RestOperations` so `OAuth2TokenRelayFilter` can refresh the token if needed.

146.2.4. Resource Server Token Relay

If your app has `@EnableResourceServer` you might want to relay the incoming token downstream to other services. If you use a `RestTemplate` to contact the downstream services then this is just a matter of how to create the template with the right context.

If your service uses `UserInfoTokenServices` to authenticate incoming tokens (i.e. it is using the `security.oauth2.user-info-uri` configuration), then you can simply create an `OAuth2RestTemplate` using an autowired `OAuth2ClientContext` (it will be populated by the authentication process before it hits the backend code). Equivalently (with Spring Boot 1.4), you could inject a `UserInfoRestTemplateFactory` and grab its `OAuth2RestTemplate` in your configuration. For example:

MyConfiguration.java

```
@Bean
public OAuth2RestTemplate restTemplate(UserInfoRestTemplateFactory factory) {
    return factory.getUserInfoRestTemplate();
}
```

This rest template will then have the same `OAuth2ClientContext` (request-scoped) that is used by the authentication filter, so you can use it to send requests with the same access token.

If your app is not using `UserInfoTokenServices` but is still a client (i.e. it declares `@EnableOAuth2Client` or `@EnableOAuth2Sso`), then with Spring Security Cloud any `OAuth2RestOperations` that the user creates from an `@Autowired` `OAuth2Context` will also forward tokens. This feature is implemented by default as an MVC handler interceptor, so it only works in Spring MVC. If you are not using MVC you could use a custom filter or AOP interceptor wrapping an `AccessTokenContextRelay` to provide the same feature.

Here's a basic example showing the use of an autowired rest template created elsewhere ("foo.com" is a Resource Server accepting the same tokens as the surrounding app):

MyController.java

```
@Autowired
private OAuth2RestOperations restTemplate;

@RequestMapping("/relay")
public String relay() {
    ResponseEntity<String> response =
      restTemplate.getForEntity("https://foo.com/bar", String.class);
    return "Success! (" + response.getBody() + ")";
}
```

If you don’t want to forward tokens (and that is a valid choice, since you might want to act as yourself, rather than the client that sent you the token), then you only need to create your own `OAuth2Context` instead of autowiring the default one.

Feign clients will also pick up an interceptor that uses the `OAuth2ClientContext` if it is available, so they should also do a token relay anywhere where a `RestTemplate` would.

Chapter 147. Configuring Authentication Downstream of a Zuul Proxy

You can control the authorization behaviour downstream of an `@EnableZuulProxy` through the `proxy.auth.*` settings. Example:

application.yml

```
proxy:
  auth:
    routes:
      customers: oauth2
      stores: passthru
      recommendations: none
```

In this example the "customers" service gets an OAuth2 token relay, the "stores" service gets a passthrough (the authorization header is just passed downstream), and the "recommendations" service has its authorization header removed. The default behaviour is to do a token relay if there is a token available, and passthru otherwise.

See ProxyAuthenticationProperties for full details.

Spring Cloud Sleuth

Adrian Cole, Spencer Gibb, Marcin Grzejszczak, Dave Syer, Jay Bryant

Hoxton.SR8

Chapter 148. Introduction

Spring Cloud Sleuth implements a distributed tracing solution for Spring Cloud.

148.1. Terminology

Spring Cloud Sleuth borrows Dapper's terminology.

Span: The basic unit of work. For example, sending an RPC is a new span, as is sending a response to an RPC. Spans are identified by a unique 64-bit ID for the span and another 64-bit ID for the trace the span is a part of. Spans also have other data, such as descriptions, timestamped events, key-value annotations (tags), the ID of the span that caused them, and process IDs (normally IP addresses).

Spans can be started and stopped, and they keep track of their timing information. Once you create a span, you must stop it at some point in the future.

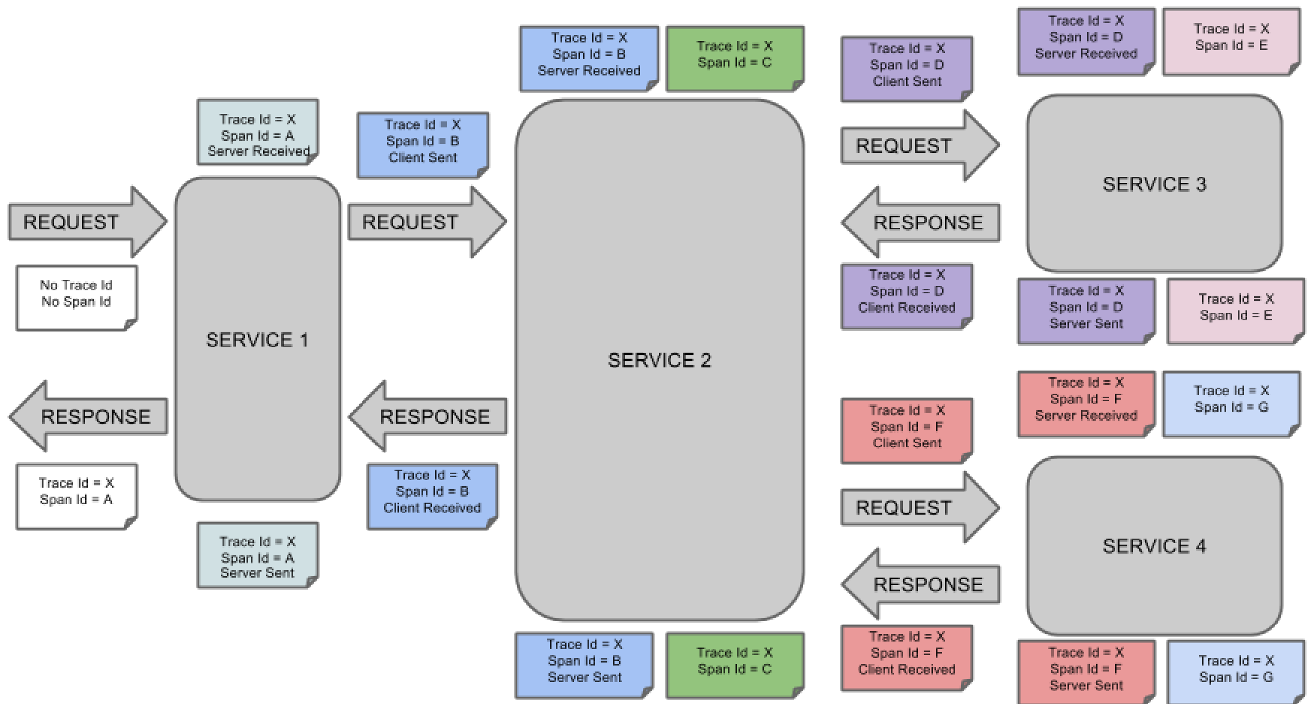
The initial span that starts a trace is called a `root span`. The value of the ID of that span is equal to the trace ID.

Trace: A set of spans forming a tree-like structure. For example, if you run a distributed big-data store, a trace might be formed by a `PUT` request.

Annotation: Used to record the existence of an event in time. With Brave instrumentation, we no longer need to set special events for Zipkin to understand who the client and server are, where the request started, and where it ended. For learning purposes, however, we mark these events to highlight what kind of an action took place.

- **cs**: Client Sent. The client has made a request. This annotation indicates the start of the span.
- **sr**: Server Received: The server side got the request and started processing it. Subtracting the `cs` timestamp from this timestamp reveals the network latency.
- **ss**: Server Sent. Annotated upon completion of request processing (when the response got sent back to the client). Subtracting the `sr` timestamp from this timestamp reveals the time needed by the server side to process the request.
- **cr**: Client Received. Signifies the end of the span. The client has successfully received the response from the server side. Subtracting the `cs` timestamp from this timestamp reveals the whole time needed by the client to receive the response from the server.

The following image shows how **Span** and **Trace** look in a system, together with the Zipkin annotations:

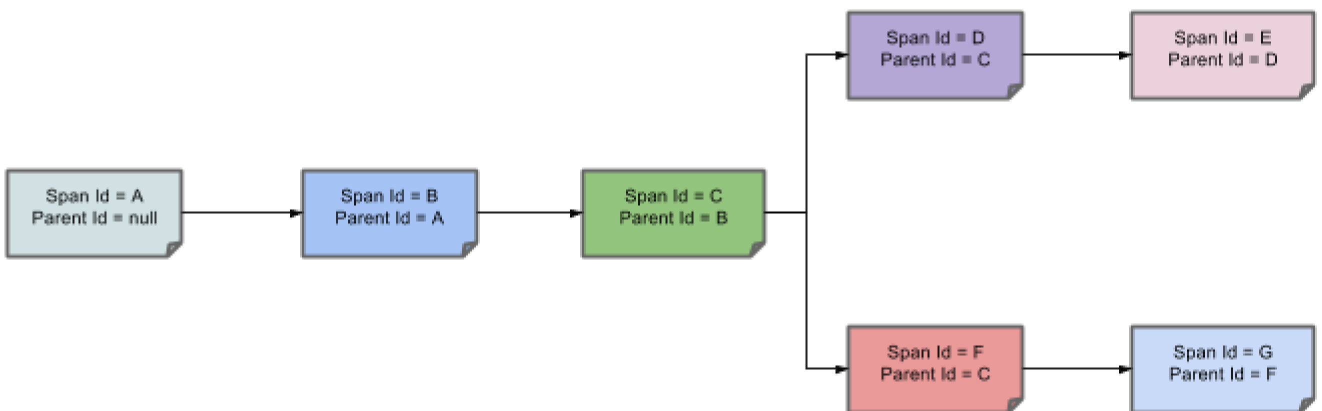


Each color of a note signifies a span (there are seven spans - from **A** to **G**). Consider the following note:

```
Trace Id = X
Span Id = D
Client Sent
```

This note indicates that the current span has **Trace Id** set to **X** and **Span Id** set to **D**. Also, the `Client Sent` event took place.

The following image shows how parent-child relationships of spans look:

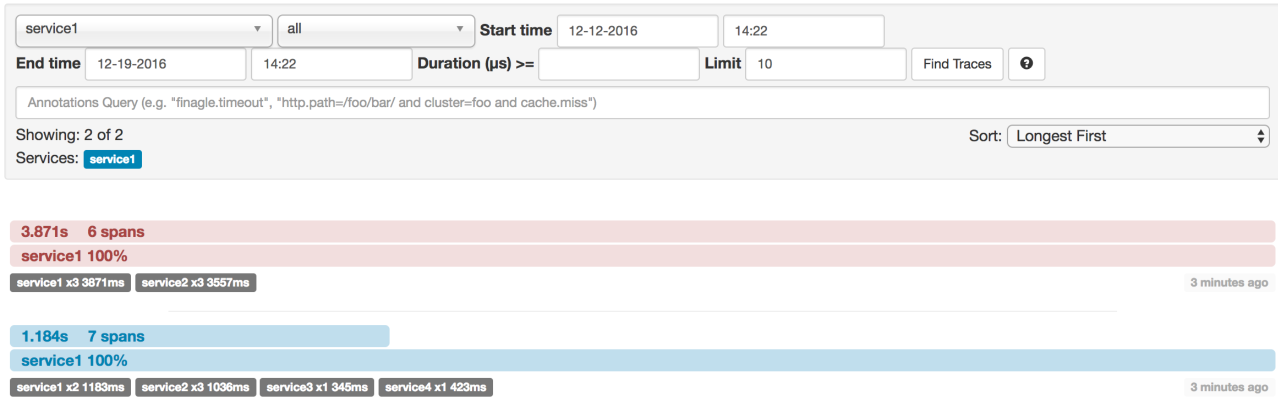


148.2. Purpose

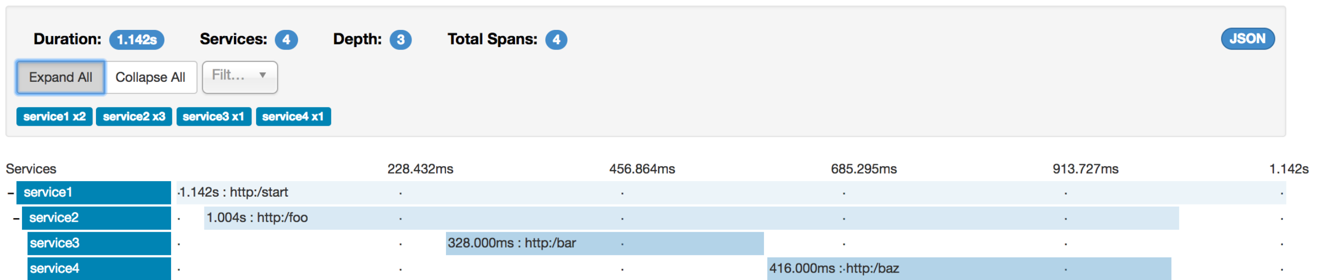
The following sections refer to the example shown in the preceding image.

148.2.1. Distributed Tracing with Zipkin

This example has seven spans. If you go to traces in Zipkin, you can see this number in the second trace, as shown in the following image:



However, if you pick a particular trace, you can see four spans, as shown in the following image:



When you pick a particular trace, you see merged spans. That means that, if there were two spans sent to Zipkin with Server Received and Server Sent or Client Received and Client Sent annotations, they are presented as a single span.

Why is there a difference between the seven and four spans in this case?

- One span comes from the `http:/start` span. It has the Server Received (`sr`) and Server Sent (`ss`) annotations.
- Two spans come from the RPC call from `service1` to `service2` to the `http:/foo` endpoint. The Client Sent (`cs`) and Client Received (`cr`) events took place on the `service1` side. Server Received (`sr`) and Server Sent (`ss`) events took place on the `service2` side. These two spans form one logical span related to an RPC call.
- Two spans come from the RPC call from `service2` to `service3` to the `http:/bar` endpoint. The Client Sent (`cs`) and Client Received (`cr`) events took place on the `service2` side. The Server Received (`sr`) and Server Sent (`ss`) events took place on the `service3` side. These two spans form one logical span related to an RPC call.
- Two spans come from the RPC call from `service2` to `service4` to the `http:/baz` endpoint. The Client Sent (`cs`) and Client Received (`cr`) events took place on the `service2` side. Server Received (`sr`) and Server Sent (`ss`) events took place on the `service4` side. These two spans form one logical span related to an RPC call.

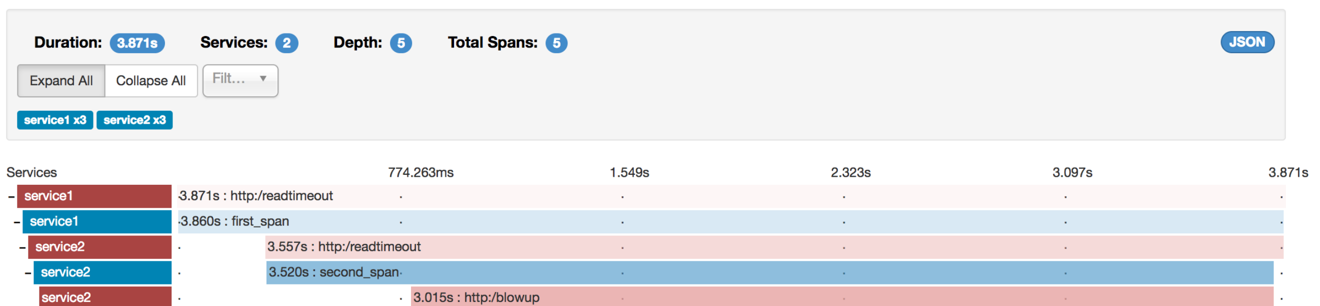
So, if we count the physical spans, we have one from `http:/start`, two from `service1` calling `service2`, two from `service2` calling `service3`, and two from `service2` calling `service4`. In sum, we have a total of seven spans.

Logically, we see the information of four total Spans because we have one span related to the incoming request to `service1` and three spans related to RPC calls.

148.2.2. Visualizing errors

Zipkin lets you visualize errors in your trace. When an exception was thrown and was not caught, we set proper tags on the span, which Zipkin can then properly colorize. You could see in the list of traces one trace that is red. That appears because an exception was thrown.

If you click that trace, you see a similar picture, as follows:



If you then click on one of the spans, you see the following

service2.http:/readtimeout: 3.557s

AKA: service1,service2

Date Time	Relative Time	Annotation	Address
19/12/2016, 14:19:23	307.000ms	Client Send	127.0.0.1:8081 (service1)
19/12/2016, 14:19:23	310.000ms	Server Receive	127.0.0.1:8082 (service2)
19/12/2016, 14:19:26	3.836s	Server Send	127.0.0.1:8082 (service2)
19/12/2016, 14:19:27	3.864s	Client Receive	127.0.0.1:8081 (service1)

Key	Value
error	Request processing failed; nested exception is org.springframework.web.client.ResourceAccessException: I/O error on GET request for "http://localhost:8082/blowup": Read timed out; nested exception is java.net.SocketTimeoutException: Read timed out
http.host	localhost
http.method	GET
http.path	/readtimeout
http.status_code	500
http.url	http://localhost:8082/readtimeout
mvc.controller.class	BasicErrorController
mvc.controller.method	error

The span shows the reason for the error and the whole stack trace related to it.

148.2.3. Distributed Tracing with Brave

Starting with version `2.0.0`, Spring Cloud Sleuth uses Brave as the tracing library. Consequently, Sleuth no longer takes care of storing the context but delegates that work to Brave.

Due to the fact that Sleuth had different naming and tagging conventions than Brave, we decided to follow Brave's conventions from now on. However, if you want to use the legacy Sleuth approaches, you can set the `spring.sleuth.http.legacy.enabled` property to `true`.

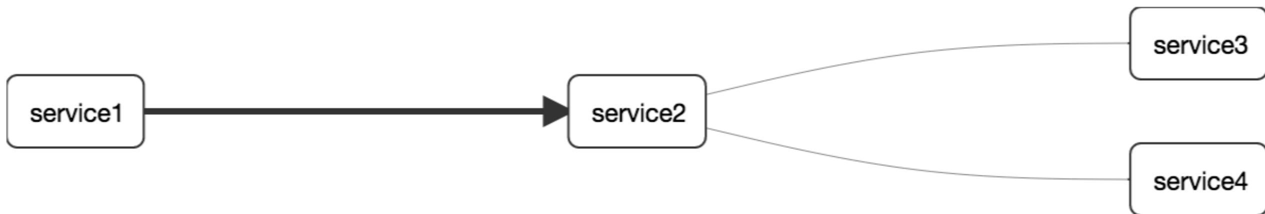
148.2.4. Live examples

Pivotal Web Services

Click the Pivotal Web Services icon to see it live!Click the Pivotal Web Services icon to see it live!

Click here to see it live!

The dependency graph in Zipkin should resemble the following image:



Pivotal Web Services

*Click the Pivotal Web Services icon to see it live!**Click the Pivotal Web Services icon to see it live!*

Click here to see it live!

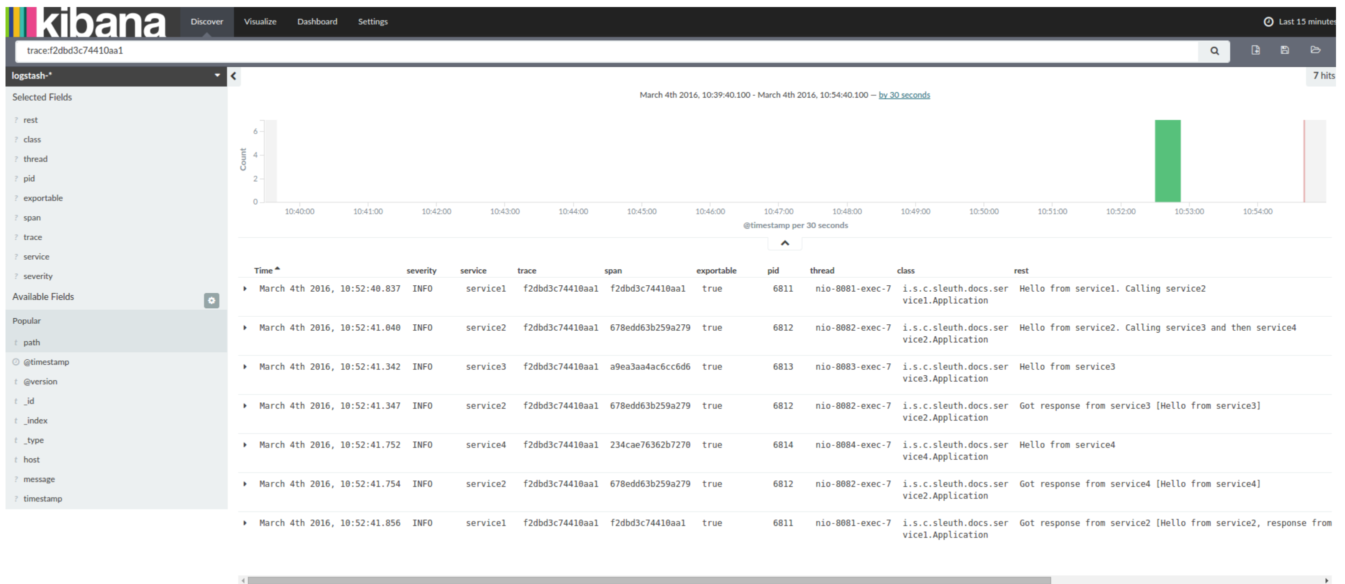
148.2.5. Log correlation

When using grep to read the logs of those four applications by scanning for a trace ID equal to (for example) `2485ec27856c56f4`, you get output resembling the following:

```
service1.log:2016-02-26 11:15:47.561  INFO
[service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1]
i.s.c.sleuth.docs.service1.Application   : Hello from service1. Calling service2
service2.log:2016-02-26 11:15:47.710  INFO
[service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1]
i.s.c.sleuth.docs.service2.Application   : Hello from service2. Calling service3 and
then service4
service3.log:2016-02-26 11:15:47.895  INFO
[service3,2485ec27856c56f4,1210be13194bfe5,true] 68060 --- [nio-8083-exec-1]
i.s.c.sleuth.docs.service3.Application   : Hello from service3
service2.log:2016-02-26 11:15:47.924  INFO
[service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1]
i.s.c.sleuth.docs.service2.Application   : Got response from service3 [Hello from
service3]
service4.log:2016-02-26 11:15:48.134  INFO
[service4,2485ec27856c56f4,1b1845262ffba49d,true] 68061 --- [nio-8084-exec-1]
i.s.c.sleuth.docs.service4.Application   : Hello from service4
service2.log:2016-02-26 11:15:48.156  INFO
[service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1]
i.s.c.sleuth.docs.service2.Application   : Got response from service4 [Hello from
service4]
service1.log:2016-02-26 11:15:48.182  INFO
[service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1]
i.s.c.sleuth.docs.service1.Application   : Got response from service2 [Hello from
service2, response from service3 [Hello from service3] and from service4 [Hello from
service4]]
```

If you use a log aggregating tool (such as Kibana, Splunk, and others), you can order the events that

took place. An example from Kibana would resemble the following image:



If you want to use Logstash, the following listing shows the Grok pattern for Logstash:

```
filter {
  # pattern matching logback pattern
  grok {
    match => { "message" =>
"%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trac
e},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---
\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }
  }
  date {
    match => ["timestamp", "ISO8601"]
  }
  mutate {
    remove_field => ["timestamp"]
  }
}
```

If you want to use Grok together with the logs from Cloud Foundry, you have to use the following pattern:

```
filter {
  # pattern matching logback pattern
  grok {
    match => { "message" =>
"(?m)OUT\s+%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%
{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---
\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }
  }
  date {
    match => ["timestamp", "ISO8601"]
  }
  mutate {
    remove_field => ["timestamp"]
  }
}
```

JSON Logback with Logstash

Often, you do not want to store your logs in a text file but in a JSON file that Logstash can immediately pick. To do so, you have to do the following (for readability, we pass the dependencies in the `groupId:artifactId:version` notation).

Dependencies Setup

1. Ensure that Logback is on the classpath (`ch.qos.logback:logback-core`).
2. Add Logstash Logback encode. For example, to use version `4.6`, add `net.logstash.logback:logstash-logback-encoder:4.6`.

Logback Setup

Consider the following example of a Logback configuration file (named logback-spring.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>

    <springProperty scope="context" name="springAppName"
source="spring.application.name"/>
    <!-- Example for logging into the build folder of your project -->
    <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>

    <!-- You can override this to have a custom pattern -->
    <property name="CONSOLE_LOG_PATTERN"
              value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){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 to log to console -->
```

```
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <!-- Minimum logging level to be presented in the console logs-->
            <level>DEBUG</level>
        </filter>
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>

    <!-- Appender to log to file -->
    <appender name="flatfile" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOG_FILE}</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>

    <!-- Appender to log to file in a JSON format -->
    <appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOG_FILE}.json</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder
class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp>
                    <timeZone>UTC</timeZone>
                </timestamp>
                <pattern>
                    <pattern>
                        {
                        "severity": "%level",
                        "service": "${springAppName:-}",
                        "trace": "%X{traceId:-}",
                        "span": "%X{spanId:-}",
                        "baggage": "%X{key:-}",
                        "pid": "${PID:-}",
                        "thread": "%thread",
                        "class": "%logger{40}",
                        "rest": "%message"
                        }
                    </pattern>
```

```
                </pattern>
            </providers>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="console"/>
        <!-- uncomment this to have also JSON logs -->
        <!--<appender-ref ref="logstash"/>-->
        <!--<appender-ref ref="flatfile"/>-->
    </root>
</configuration>
```

That Logback configuration file:

- Logs information from the application in a JSON format to a `build/${spring.application.name}.json` file.
- Has commented out two additional appenders: console and standard log file.
- Has the same logging pattern as the one presented in the previous section.

If you use a custom `logback-spring.xml`, you must pass the `spring.application.name` in the `bootstrap` rather than the `application` property file. Otherwise, your custom logback file does not properly read the property.

148.2.6. Propagating Span Context

The span context is the state that must get propagated to any child spans across process boundaries. Part of the Span Context is the Baggage. The trace and span IDs are a required part of the span context. Baggage is an optional part.

Baggage is a set of key:value pairs stored in the span context. Baggage travels together with the trace and is attached to every span. Spring Cloud Sleuth understands that a header is baggage-related if the HTTP header is prefixed with `baggage-` and, for messaging, it starts with `baggage_`.

There is currently no limitation of the count or size of baggage items. However, keep in mind that too many can decrease system throughput or increase RPC latency. In extreme cases, too much baggage can crash the application, due to exceeding transport-level message or header capacity.

The following example shows setting baggage on a span:

```
Span initialSpan = this.tracer.nextSpan().name("span").start();
ExtraFieldPropagation.set(initialSpan.context(), "foo", "bar");
ExtraFieldPropagation.set(initialSpan.context(), "UPPER_CASE", "someValue");
```

Baggage versus Span Tags

Baggage travels with the trace (every child span contains the baggage of its parent). Zipkin has no knowledge of baggage and does not receive that information.

Starting from Sleuth 2.0.0 you have to pass the baggage key names explicitly in your project configuration. Read more about that setup here

Tags are attached to a specific span. In other words, they are presented only for that particular span. However, you can search by tag to find the trace, assuming a span having the searched tag value exists.

If you want to be able to lookup a span based on baggage, you should add a corresponding entry as a tag in the root span.

The span must be in scope.

The following listing shows integration tests that use baggage:

The setup

```
spring.sleuth:
  baggage-keys:
    - baz
    - bizarrecase
  propagation-keys:
    - foo
    - upper_case
```

The code

```
initialSpan.tag("foo",
        ExtraFieldPropagation.get(initialSpan.context(), "foo"));
initialSpan.tag("UPPER_CASE",
        ExtraFieldPropagation.get(initialSpan.context(), "UPPER_CASE"));
```

148.3. Adding Sleuth to the Project

This section addresses how to add Sleuth to your project with either Maven or Gradle.

To ensure that your application name is properly displayed in Zipkin, set the `spring.application.name` property in `bootstrap.yml`.

148.3.1. Only Sleuth (log correlation)

If you want to use only Spring Cloud Sleuth without the Zipkin integration, add the `spring-cloud-starter-sleuth` module to your project.

The following example shows how to add Sleuth with Maven:

Maven

```
<dependencyManagement> ①
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train.version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
</dependencyManagement>

<dependency> ②
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.

② Add the dependency to `spring-cloud-starter-sleuth`.

The following example shows how to add Sleuth with Gradle:

Gradle

```
dependencyManagement { ①
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
    }
}

dependencies { ②
    compile "org.springframework.cloud:spring-cloud-starter-sleuth"
}
```

① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.

② Add the dependency to `spring-cloud-starter-sleuth`.

148.3.2. Sleuth with Zipkin via HTTP

If you want both Sleuth and Zipkin, add the `spring-cloud-starter-zipkin` dependency.

The following example shows how to do so for Maven:

Maven

```
<dependencyManagement> ①
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train.version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
</dependencyManagement>

<dependency> ②
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.

② Add the dependency to `spring-cloud-starter-zipkin`.

The following example shows how to do so for Gradle:

Gradle

```
dependencyManagement { ①
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
    }
}

dependencies { ②
    compile "org.springframework.cloud:spring-cloud-starter-zipkin"
}
```

① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.

② Add the dependency to `spring-cloud-starter-zipkin`.

148.3.3. Sleuth with Zipkin over RabbitMQ or Kafka

If you want to use RabbitMQ or Kafka instead of HTTP, add the `spring-rabbit` or `spring-kafka` dependency. The default destination name is `zipkin`.

If using Kafka, you must set the property `spring.zipkin.sender.type` property accordingly:

```
spring.zipkin.sender.type: kafka
```

spring-cloud-sleuth-stream is deprecated and incompatible with these destinations.

If you want Sleuth over RabbitMQ, add the spring-cloud-starter-zipkin and spring-rabbit dependencies.

The following example shows how to do so for Gradle:

Maven

```
<dependencyManagement> ①
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train.version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
</dependencyManagement>

<dependency> ②
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency> ③
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
</dependency>
```

① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.

② Add the dependency to spring-cloud-starter-zipkin. That way, all nested dependencies get downloaded.

③ To automatically configure RabbitMQ, add the spring-rabbit dependency.

Gradle

```
dependencyManagement { ①
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    compile "org.springframework.cloud:spring-cloud-starter-zipkin" ②
    compile "org.springframework.amqp:spring-rabbit" ③
}
```

① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.

② Add the dependency to `spring-cloud-starter-zipkin`. That way, all nested dependencies get downloaded.

③ To automatically configure RabbitMQ, add the `spring-rabbit` dependency.

148.4. Overriding the auto-configuration of Zipkin

Spring Cloud Sleuth supports sending traces to multiple tracing systems as of version 2.1.0. In order to get this to work, every tracing system needs to have a `Reporter<Span>` and `Sender`. If you want to override the provided beans you need to give them a specific name. To do this you can use respectively `ZipkinAutoConfiguration.REPORTER_BEAN_NAME` and `ZipkinAutoConfiguration.SENDER_BEAN_NAME`.

```
@Configuration
protected static class MyConfig {

    @Bean(ZipkinAutoConfiguration.REPORTER_BEAN_NAME)
    Reporter<zipkin2.Span> myReporter() {
        return AsyncReporter.create(mySender());
    }

    @Bean(ZipkinAutoConfiguration.SENDER_BEAN_NAME)
    MySender mySender() {
        return new MySender();
    }

    static class MySender extends Sender {

        private boolean spanSent = false;

        boolean isSpanSent() {
            return this.spanSent;
        }

        @Override
        public Encoding encoding() {
            return Encoding.JSON;
        }

        @Override
        public int messageMaxBytes() {
            return Integer.MAX_VALUE;
        }

        @Override
        public int messageSizeInBytes(List<byte[]> encodedSpans) {
            return encoding().listSizeInBytes(encodedSpans);
        }

        @Override
        public Call<Void> sendSpans(List<byte[]> encodedSpans) {
            this.spanSent = true;
            return Call.create(null);
        }

    }

}
```

Chapter 149. Additional Resources

You can watch a video of Reshmi Krishna and Marcin Grzejszczak talking about Spring Cloud Sleuth and Zipkin by clicking here.

You can check different setups of Sleuth and Brave in the openzipkin/sleuth-webmvc-example repository.

Chapter 150. Features

- Adds trace and span IDs to the Slf4J MDC, so you can extract all the logs from a given trace or span in a log aggregator, as shown in the following example logs:

  ```
  2016-02-02 15:30:57.902  INFO [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030
  --- [nio-8081-exec-3] ...
  2016-02-02 15:30:58.372 ERROR [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030
  --- [nio-8081-exec-3] ...
  2016-02-02 15:31:01.936  INFO [bar,46ab0d418373cbc9,46ab0d418373cbc9,false] 23030
  --- [nio-8081-exec-4] ...
  ```

  Notice the `[appname,traceId,spanId,exportable]` entries from the MDC:

  - **`spanId`**: The ID of a specific operation that took place.
  - **`appname`**: The name of the application that logged the span.
  - **`traceId`**: The ID of the latency graph that contains the span.
  - **`exportable`**: Whether the log should be exported to Zipkin. When would you like the span not to be exportable? When you want to wrap some operation in a Span and have it written to the logs only.
- Provides an abstraction over common distributed tracing data models: traces, spans (forming a DAG), annotations, and key-value annotations. Spring Cloud Sleuth is loosely based on HTrace but is compatible with Zipkin (Dapper).
- Sleuth records timing information to aid in latency analysis. By using sleuth, you can pinpoint causes of latency in your applications.
- Sleuth is written to not log too much and to not cause your production application to crash. To that end, Sleuth:
  - Propagates structural data about your call graph in-band and the rest out-of-band.
  - Includes opinionated instrumentation of layers such as HTTP.
  - Includes a sampling policy to manage volume.
  - Can report to a Zipkin system for query and visualization.
- Instruments common ingress and egress points from Spring applications (servlet filter, async endpoints, rest template, scheduled actions, message channels, Zuul filters, and Feign client).
- Sleuth includes default logic to join a trace across HTTP or messaging boundaries. For example, HTTP propagation works over Zipkin-compatible request headers.
- Sleuth can propagate context (also known as baggage) between processes. Consequently, if you set a baggage element on a Span, it is sent downstream to other processes over either HTTP or messaging.
- Provides a way to create or continue spans and add tags and logs through annotations.
- If `spring-cloud-sleuth-zipkin` is on the classpath, the app generates and collects Zipkin-compatible traces. By default, it sends them over HTTP to a Zipkin server on localhost (port

9411). You can configure the location of the service by setting `spring.zipkin.baseUrl`.

  - If you depend on `spring-rabbit`, your app sends traces to a RabbitMQ broker instead of HTTP.
  - If you depend on `spring-kafka`, and set `spring.zipkin.sender.type: kafka`, your app sends traces to a Kafka broker instead of HTTP.

`spring-cloud-sleuth-stream` is deprecated and should no longer be used.

- Spring Cloud Sleuth is OpenTracing compatible.

The SLF4J MDC is always set and logback users immediately see the trace and span IDs in logs per the example shown earlier. Other logging systems have to configure their own formatter to get the same result. The default is as follows: `logging.pattern.level` set to `%5p [${spring.zipkin.service.name:${spring.application.name:-}},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}]` (this is a Spring Boot feature for logback users). If you do not use SLF4J, this pattern is NOT automatically applied.

150.1. Introduction to Brave

Starting with version `2.0.0`, Spring Cloud Sleuth uses Brave as the tracing library. For your convenience, we embed part of the Brave's docs here.

In the vast majority of cases you need to just use the `Tracer` or `SpanCustomizer` beans from Brave that Sleuth provides. The documentation below contains a high overview of what Brave is and how it works.

Brave is a library used to capture and report latency information about distributed operations to Zipkin. Most users do not use Brave directly. They use libraries or frameworks rather than employ Brave on their behalf.

This module includes a tracer that creates and joins spans that model the latency of potentially distributed work. It also includes libraries to propagate the trace context over network boundaries (for example, with HTTP headers).

150.1.1. Tracing

Most importantly, you need a `brave.Tracer`, configured to report to Zipkin.

The following example setup sends trace data (spans) to Zipkin over HTTP (as opposed to Kafka):

```
class MyClass {

    private final Tracer tracer;

    // Tracer will be autowired
    MyClass(Tracer tracer) {
        this.tracer = tracer;
    }

    void doSth() {
        Span span = tracer.newTrace().name("encode").start();
        // ...
    }
}
```

If your span contains a name longer than 50 chars, then that name is truncated to 50 chars. Your names have to be explicit and concrete. Big names lead to latency issues and sometimes even thrown exceptions.

The tracer creates and joins spans that model the latency of potentially distributed work. It can employ sampling to reduce overhead during the process, to reduce the amount of data sent to Zipkin, or both.

Spans returned by a tracer report data to Zipkin when finished or do nothing if unsampled. After starting a span, you can annotate events of interest or add tags containing details or lookup keys.

Spans have a context that includes trace identifiers that place the span at the correct spot in the tree representing the distributed operation.

150.1.2. Local Tracing

When tracing code that never leaves your process, run it inside a scoped span.

```
@Autowired Tracer tracer;

// Start a new trace or a span within an existing trace representing an operation
ScopedSpan span = tracer.startScopedSpan("encode");
try {
  // The span is in "scope" meaning downstream code such as loggers can see trace IDs
  return encoder.encode();
} catch (RuntimeException | Error e) {
  span.error(e); // Unless you handle exceptions, you might not know the operation
failed!
  throw e;
} finally {
  span.finish(); // always finish the span
}
```

When you need more features, or finer control, use the `Span` type:

```
@Autowired Tracer tracer;

// Start a new trace or a span within an existing trace representing an operation
Span span = tracer.nextSpan().name("encode").start();
// Put the span in "scope" so that downstream code such as loggers can see trace IDs
try (SpanInScope ws = tracer.withSpanInScope(span)) {
  return encoder.encode();
} catch (RuntimeException | Error e) {
  span.error(e); // Unless you handle exceptions, you might not know the operation
failed!
  throw e;
} finally {
  span.finish(); // note the scope is independent of the span. Always finish a span.
}
```

Both of the above examples report the exact same span on finish!

In the above example, the span will be either a new root span or the next child in an existing trace.

150.1.3. Customizing Spans

Once you have a span, you can add tags to it. The tags can be used as lookup keys or details. For example, you might add a tag with your runtime version, as shown in the following example:

```
span.tag("clnt/finagle.version", "6.36.0");
```

When exposing the ability to customize spans to third parties, prefer `brave.SpanCustomizer` as opposed to `brave.Span`. The former is simpler to understand and test and does not tempt users with span lifecycle hooks.

```
interface MyTraceCallback {
  void request(Request request, SpanCustomizer customizer);
}
```

Since `brave.Span` implements `brave.SpanCustomizer`, you can pass it to users, as shown in the following example:

```
for (MyTraceCallback callback : userCallbacks) {
  callback.request(request, span);
}
```

150.1.4. Implicitly Looking up the Current Span

Sometimes, you do not know if a trace is in progress or not, and you do not want users to do null checks. `brave.CurrentSpanCustomizer` handles this problem by adding data to any span that's in progress or drops, as shown in the following example:

Ex.

```
// The user code can then inject this without a chance of it being null.
@Autowired SpanCustomizer span;

void userCode() {
  span.annotate("tx.started");
  ...
}
```

150.1.5. RPC tracing

Check for instrumentation written here and Zipkin's list before rolling your own RPC instrumentation.

RPC tracing is often done automatically by interceptors. Behind the scenes, they add tags and events that relate to their role in an RPC operation.

The following example shows how to add a client span:

```java
@Autowired Tracing tracing;
@Autowired Tracer tracer;

// before you send a request, add metadata that describes the operation
span = tracer.nextSpan().name(service + "/" + method).kind(CLIENT);
span.tag("myrpc.version", "1.0.0");
span.remoteServiceName("backend");
span.remoteIpAndPort("172.3.4.1", 8108);

// Add the trace context to the request, so it can be propagated in-band
tracing.propagation().injector(Request::addHeader)
                     .inject(span.context(), request);

// when the request is scheduled, start the span
span.start();

// if there is an error, tag the span
span.tag("error", error.getCode());
// or if there is an exception
span.error(exception);

// when the response is complete, finish the span
span.finish();
```

One-Way tracing

Sometimes, you need to model an asynchronous operation where there is a request but no response. In normal RPC tracing, you use `span.finish()` to indicate that the response was received. In one-way tracing, you use `span.flush()` instead, as you do not expect a response.

The following example shows how a client might model a one-way operation:

```java
@Autowired Tracing tracing;
@Autowired Tracer tracer;

// start a new span representing a client request
oneWaySend = tracer.nextSpan().name(service + "/" + method).kind(CLIENT);

// Add the trace context to the request, so it can be propagated in-band
tracing.propagation().injector(Request::addHeader)
                     .inject(oneWaySend.context(), request);

// fire off the request asynchronously, totally dropping any response
request.execute();

// start the client side and flush instead of finish
oneWaySend.start().flush();
```

The following example shows how a server might handle a one-way operation:

```
@Autowired Tracing tracing;
@Autowired Tracer tracer;

// pull the context out of the incoming request
extractor = tracing.propagation().extractor(Request::getHeader);

// convert that context to a span which you can name and add tags to
oneWayReceive = nextSpan(tracer, extractor.extract(request))
    .name("process-request")
    .kind(SERVER)
    ... add tags etc.

// start the server side and flush instead of finish
oneWayReceive.start().flush();

// you should not modify this span anymore as it is complete. However,
// you can create children to represent follow-up work.
next = tracer.newSpan(oneWayReceive.context()).name("step2").start();
```

Chapter 151. Sampling

Sampling may be employed to reduce the data collected and reported out of process. When a span is not sampled, it adds no overhead (a noop).

Sampling is an up-front decision, meaning that the decision to report data is made at the first operation in a trace and that decision is propagated downstream.

By default, a global sampler applies a single rate to all traced operations. `Tracer.Builder.sampler` controls this setting, and it defaults to tracing every request.

151.1. Declarative sampling

Some applications need to sample based on the type or annotations of a java method.

Most users use a framework interceptor to automate this sort of policy. The following example shows how that might work internally:

```
@Autowired Tracer tracer;

// derives a sample rate from an annotation on a java method
DeclarativeSampler<Traced> sampler = DeclarativeSampler.create(Traced::sampleRate);

@Around("@annotation(traced)")
public Object traceThing(ProceedingJoinPoint pjp, Traced traced) throws Throwable {
  // When there is no trace in progress, this decides using an annotation
  Sampler decideUsingAnnotation = declarativeSampler.toSampler(traced);
  Tracer tracer = tracer.withSampler(decideUsingAnnotation);

  // This code looks the same as if there was no declarative override
  ScopedSpan span = tracer.startScopedSpan(spanName(pjp));
  try {
    return pjp.proceed();
  } catch (RuntimeException | Error e) {
    span.error(e);
    throw e;
  } finally {
    span.finish();
  }
}
```

151.2. Custom sampling

Depending on what the operation is, you may want to apply different policies. For example, you might not want to trace requests to static resources such as images, or you might want to trace all requests to a new api.

Most users use a framework interceptor to automate this sort of policy. The following example

shows how that might work internally:

```
@Autowired Tracer tracer;
@Autowired Sampler fallback;

Span nextSpan(final Request input) {
  Sampler requestBased = Sampler() {
    @Override public boolean isSampled(long traceId) {
      if (input.url().startsWith("/experimental")) {
        return true;
      } else if (input.url().startsWith("/static")) {
        return false;
      }
      return fallback.isSampled(traceId);
    }
  };
  return tracer.withSampler(requestBased).nextSpan();
}
```

151.3. Sampling in Spring Cloud Sleuth

Sampling only applies to tracing backends, such as Zipkin. Trace IDs appear in logs regardless of sample rate. Sampling is a way to prevent overloading the system, by consistently tracing some, but not all requests.

The default rate of 10 traces per second is controlled by the `spring.sleuth.sampler.rate` property and applies when we know Sleuth is used for reasons besides logging. Use a rate above 100 traces per second with extreme caution as it can overload your tracing system.

The sampler can be set by Java Config also, as shown in the following example:

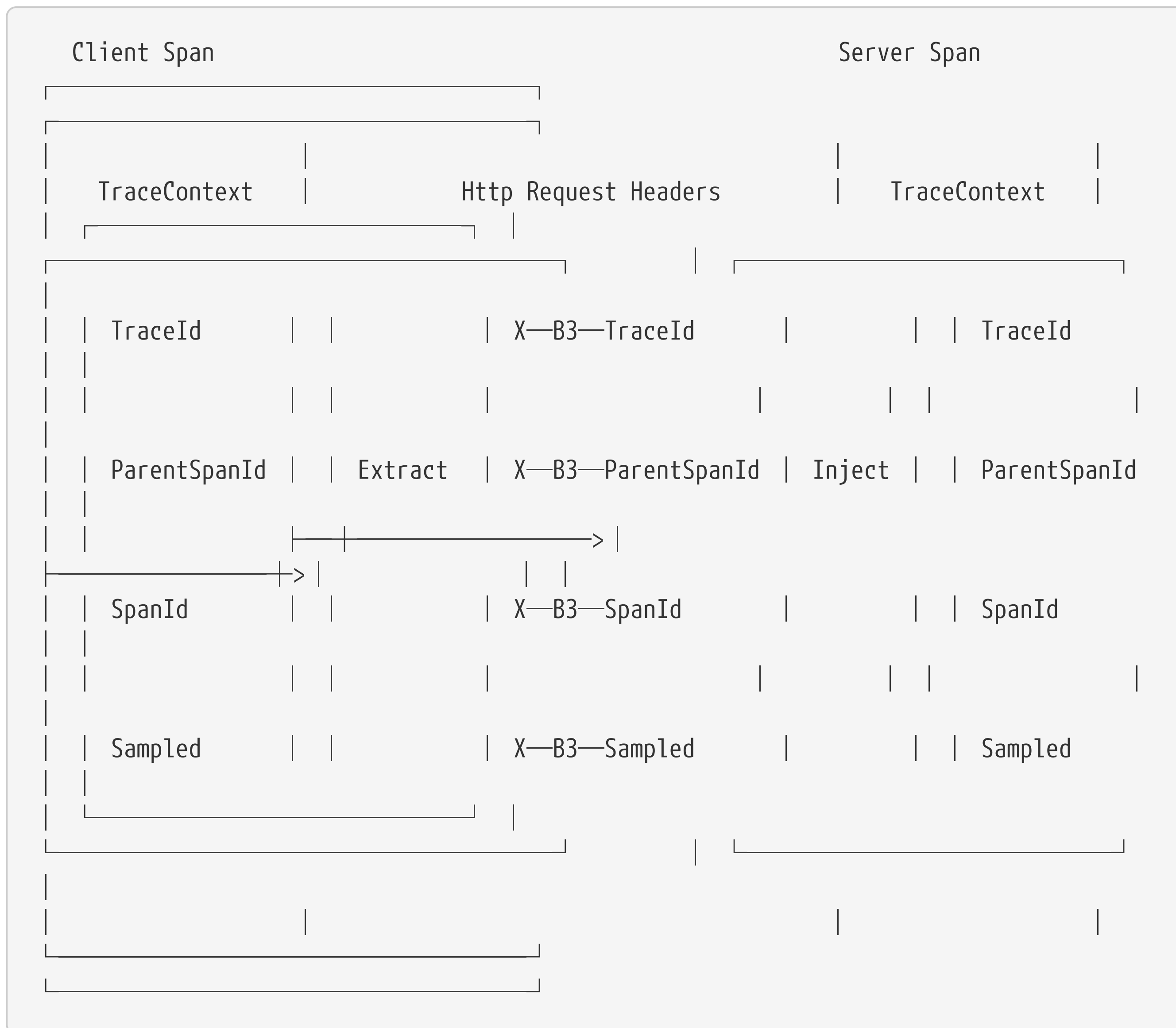
```
@Bean
public Sampler defaultSampler() {
    return Sampler.ALWAYS_SAMPLE;
}
```

You can set the HTTP header `X-B3-Flags` to `1`, or, when doing messaging, you can set the `spanFlags` header to `1`. Doing so forces the corresponding trace to be sampled regardless of the sampling configuration.

Chapter 152. Propagation

Propagation is needed to ensure activities originating from the same root are collected together in the same trace. The most common propagation approach is to copy a trace context from a client by sending an RPC request to a server receiving it.

For example, when a downstream HTTP call is made, its trace context is encoded as request headers and sent along with it, as shown in the following image:



The names above are from B3 Propagation, which is built-in to Brave and has implementations in many languages and frameworks.

Most users use a framework interceptor to automate propagation. The next two examples show how that might work for a client and a server.

The following example shows how client-side propagation might work:

```
@Autowired Tracing tracing;

// configure a function that injects a trace context into a request
injector = tracing.propagation().injector(Request.Builder::addHeader);

// before a request is sent, add the current span's context to it
injector.inject(span.context(), request);
```

The following example shows how server-side propagation might work:

```
@Autowired Tracing tracing;
@Autowired Tracer tracer;

// configure a function that extracts the trace context from a request
extractor = tracing.propagation().extractor(Request::getHeader);

// when a server receives a request, it joins or starts a new trace
span = tracer.nextSpan(extractor.extract(request));
```

152.1. Propagating extra fields

Sometimes you need to propagate extra fields, such as a request ID or an alternate trace context. For example, if you are in a Cloud Foundry environment, you might want to pass the request ID, as shown in the following example:

```
// when you initialize the builder, define the extra field you want to propagate
Tracing.newBuilder().propagationFactory(
  ExtraFieldPropagation.newFactory(B3Propagation.FACTORY, "x-vcap-request-id")
);

// later, you can tag that request ID or use it in log correlation
requestId = ExtraFieldPropagation.get("x-vcap-request-id");
```

You may also need to propagate a trace context that you are not using. For example, you may be in an Amazon Web Services environment but not be reporting data to X-Ray. To ensure X-Ray can co-exist correctly, pass-through its tracing header, as shown in the following example:

```
tracingBuilder.propagationFactory(
  ExtraFieldPropagation.newFactory(B3Propagation.FACTORY, "x-amzn-trace-id")
);
```

In Spring Cloud Sleuth all elements of the tracing builder `Tracing.newBuilder()` are defined as beans. So if you want to pass a custom `PropagationFactory`, it's enough for you to create a bean of that type and we will set it in the `Tracing` bean.

152.1.1. Prefixed fields

If they follow a common pattern, you can also prefix fields. The following example shows how to propagate `x-vcap-request-id` the field as-is but send the `country-code` and `user-id` fields on the wire as `x-baggage-country-code` and `x-baggage-user-id`, respectively:

```
Tracing.newBuilder().propagationFactory(
  ExtraFieldPropagation.newFactoryBuilder(B3Propagation.FACTORY)
                       .addField("x-vcap-request-id")
                       .addPrefixedFields("x-baggage-", Arrays.asList("country-code",
"user-id"))
                       .build()
);
```

Later, you can call the following code to affect the country code of the current trace context:

```
ExtraFieldPropagation.set("x-country-code", "FO");
String countryCode = ExtraFieldPropagation.get("x-country-code");
```

Alternatively, if you have a reference to a trace context, you can use it explicitly, as shown in the following example:

```
ExtraFieldPropagation.set(span.context(), "x-country-code", "FO");
String countryCode = ExtraFieldPropagation.get(span.context(), "x-country-code");
```

A difference from previous versions of Sleuth is that, with Brave, you must pass the list of baggage keys. There are the following properties to achieve this. With the `spring.sleuth.baggage-keys`, you set keys that get prefixed with `baggage-` for HTTP calls and `baggage_` for messaging. You can also use the `spring.sleuth.propagation-keys` property to pass a list of prefixed keys that are propagated to remote services without any prefix. You can also use the `spring.sleuth.local-keys` property to pass a list keys that will be propagated locally but will not be propagated over the wire. Notice that there's no `x-` in front of the header keys.

In order to automatically set the baggage values to Slf4j's MDC, you have to set the `spring.sleuth.log.slf4j.whitelisted-mdc-keys` property with a list of whitelisted baggage and propagation keys. E.g. `spring.sleuth.log.slf4j.whitelisted-mdc-keys=foo` will set the value of the `foo` baggage into MDC.

Remember that adding entries to MDC can drastically decrease the performance of your application!

If you want to add the baggage entries as tags, to make it possible to search for spans via the baggage entries, you can set the value of `spring.sleuth.propagation.tag.whitelisted-keys` with a list

of whitelisted baggage keys. To disable the feature you have to pass the `spring.sleuth.propagation.tag.enabled=false` property.

152.1.2. Extracting a Propagated Context

The `TraceContext.Extractor<C>` reads trace identifiers and sampling status from an incoming request or message. The carrier is usually a request object or headers.

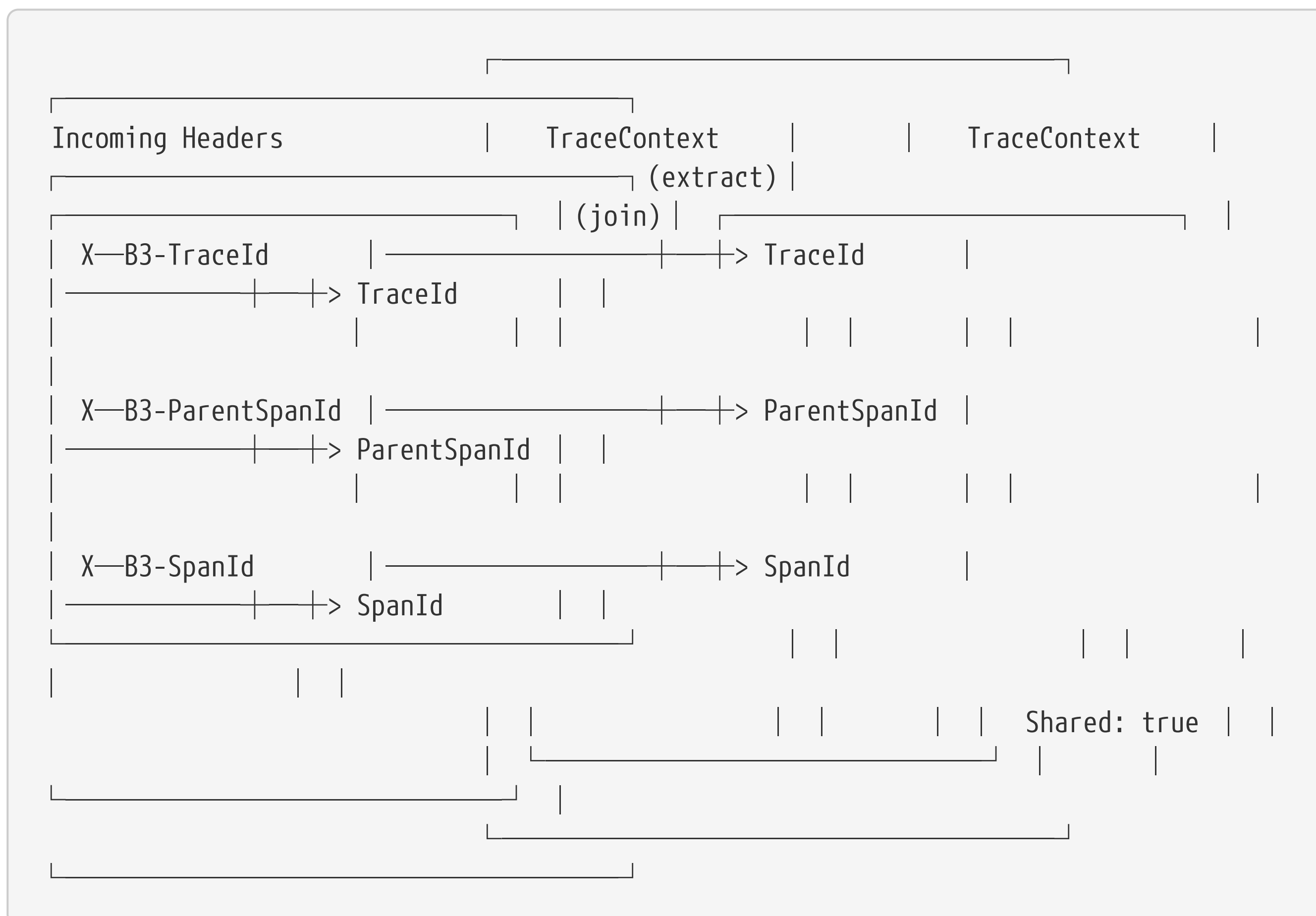
This utility is used in standard instrumentation (such as `HttpServerHandler`) but can also be used for custom RPC or messaging code.

`TraceContextOrSamplingFlags` is usually used only with `Tracer.nextSpan(extracted)`, unless you are sharing span IDs between a client and a server.

152.1.3. Sharing span IDs between Client and Server

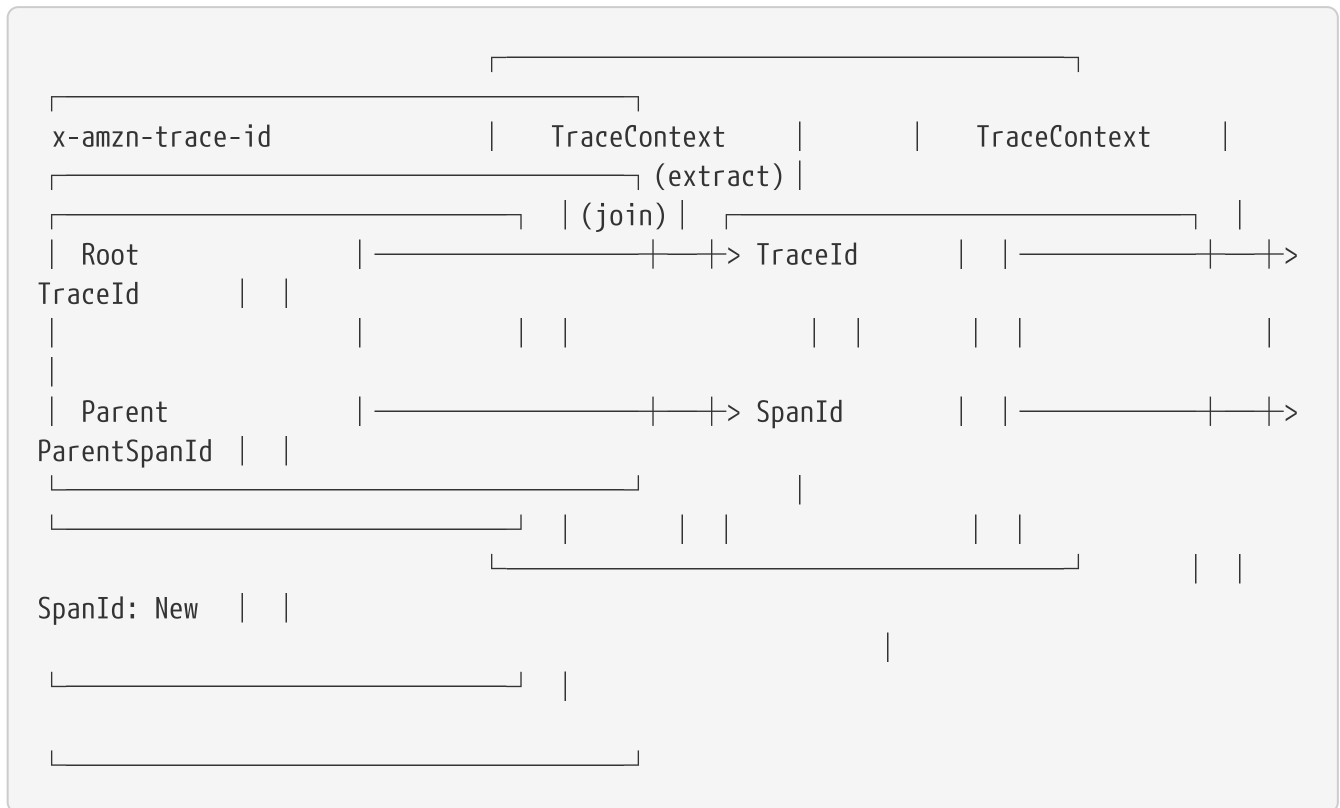
A normal instrumentation pattern is to create a span representing the server side of an RPC. `Extractor.extract` might return a complete trace context when applied to an incoming client request. `Tracer.joinSpan` attempts to continue this trace, using the same span ID if supported or creating a child span if not. When the span ID is shared, the reported data includes a flag saying so.

The following image shows an example of B3 propagation:



Some propagation systems forward only the parent span ID, detected when `Propagation.Factory.supportsJoin() == false`. In this case, a new span ID is always provisioned, and the incoming context determines the parent ID.

The following image shows an example of AWS propagation:



Note: Some span reporters do not support sharing span IDs. For example, if you set `Tracing.Builder.spanReporter(amazonXrayOrGoogleStackdrive)`, you should disable join by setting `Tracing.Builder.supportsJoin(false)`. Doing so forces a new child span on `Tracer.joinSpan()`.

152.1.4. Implementing Propagation

`TraceContext.Extractor<C>` is implemented by a `Propagation.Factory` plugin. Internally, this code creates the union type, `TraceContextOrSamplingFlags`, with one of the following:

- `TraceContext` if trace and span IDs were present.
- `TraceIdContext` if a trace ID was present but span IDs were not present.
- `SamplingFlags` if no identifiers were present.

Some `Propagation` implementations carry extra data from the point of extraction (for example, reading incoming headers) to injection (for example, writing outgoing headers). For example, it might carry a request ID. When implementations have extra data, they handle it as follows:

- If a `TraceContext` were extracted, add the extra data as `TraceContext.extra()`.
- Otherwise, add it as `TraceContextOrSamplingFlags.extra()`, which `Tracer.nextSpan` handles.

Chapter 153. Current Tracing Component

Brave supports a “current tracing component” concept, which should only be used when you have no other way to get a reference. This was made for JDBC connections, as they often initialize prior to the tracing component.

The most recent tracing component instantiated is available through `Tracing.current()`. You can also use `Tracing.currentTracer()` to get only the tracer. If you use either of these methods, do not cache the result. Instead, look them up each time you need them.

Chapter 154. Current Span

Brave supports a "current span" concept which represents the in-flight operation. You can use `Tracer.currentSpan()` to add custom tags to a span and `Tracer.nextSpan()` to create a child of whatever is in-flight.

In Sleuth, you can autowire the `Tracer` bean to retrieve the current span via `tracer.currentSpan()` method. To retrieve the current context just call `tracer.currentSpan().context()`. To get the current trace id as String you can use the `traceIdString()` method like this: `tracer.currentSpan().context().traceIdString()`.

154.1. Setting a span in scope manually

When writing new instrumentation, it is important to place a span you created in scope as the current span. Not only does doing so let users access it with `Tracer.currentSpan()`, but it also allows customizations such as SLF4J MDC to see the current trace IDs.

`Tracer.withSpanInScope(Span)` facilitates this and is most conveniently employed by using the try-with-resources idiom. Whenever external code might be invoked (such as proceeding an interceptor or otherwise), place the span in scope, as shown in the following example:

```
@Autowired Tracer tracer;

try (SpanInScope ws = tracer.withSpanInScope(span)) {
  return inboundRequest.invoke();
} finally { // note the scope is independent of the span
  span.finish();
}
```

In edge cases, you may need to clear the current span temporarily (for example, launching a task that should not be associated with the current request). To do tso, pass null to `withSpanInScope`, as shown in the following example:

```
@Autowired Tracer tracer;

try (SpanInScope cleared = tracer.withSpanInScope(null)) {
  startBackgroundThread();
}
```

Chapter 155. Instrumentation

Spring Cloud Sleuth automatically instruments all your Spring applications, so you should not have to do anything to activate it. The instrumentation is added by using a variety of technologies according to the stack that is available. For example, for a servlet web application, we use a `Filter`, and, for Spring Integration, we use `ChannelInterceptors`.

You can customize the keys used in span tags. To limit the volume of span data, an HTTP request is, by default, tagged only with a handful of metadata, such as the status code, the host, and the URL. You can add request headers by configuring `spring.sleuth.keys.http.headers` (a list of header names).

Tags are collected and exported only if there is a `Sampler` that allows it. By default, there is no such `Sampler`, to ensure that there is no danger of accidentally collecting too much data without configuring something).

Chapter 156. Span lifecycle

You can do the following operations on the Span by means of `brave.Tracer`:

- start: When you start a span, its name is assigned and the start timestamp is recorded.
- close: The span gets finished (the end time of the span is recorded) and, if the span is sampled, it is eligible for collection (for example, to Zipkin).
- continue: A new instance of span is created. It is a copy of the one that it continues.
- detach: The span does not get stopped or closed. It only gets removed from the current thread.
- create with explicit parent: You can create a new span and set an explicit parent for it.

Spring Cloud Sleuth creates an instance of `Tracer` for you. In order to use it, you can autowire it.

156.1. Creating and finishing spans

You can manually create spans by using the `Tracer`, as shown in the following example:

```
// Start a span. If there was a span present in this thread it will become
// the `newSpan`'s parent.
Span newSpan = this.tracer.nextSpan().name("calculateTax");
try (Tracer.SpanInScope ws = this.tracer.withSpanInScope(newSpan.start())) {
    // ...
    // You can tag a span
    newSpan.tag("taxValue", taxValue);
    // ...
    // You can log an event on a span
    newSpan.annotate("taxCalculated");
}
finally {
    // Once done remember to finish the span. This will allow collecting
    // the span to send it to Zipkin
    newSpan.finish();
}
```

In the preceding example, we could see how to create a new instance of the span. If there is already a span in this thread, it becomes the parent of the new span.

Always clean after you create a span. Also, always finish any span that you want to send to Zipkin.

If your span contains a name greater than 50 chars, that name is truncated to 50 chars. Your names have to be explicit and concrete. Big names lead to latency issues and sometimes even exceptions.

156.2. Continuing Spans

Sometimes, you do not want to create a new span but you want to continue one. An example of such a situation might be as follows:

- **AOP**: If there was already a span created before an aspect was reached, you might not want to create a new span.
- **Hystrix**: Executing a Hystrix command is most likely a logical part of the current processing. It is in fact merely a technical implementation detail that you would not necessarily want to reflect in tracing as a separate being.

To continue a span, you can use `brave.Tracer`, as shown in the following example:

```
// let's assume that we're in a thread Y and we've received
// the `initialSpan` from thread X
Span continuedSpan = this.tracer.toSpan(newSpan.context());
try {
    // ...
    // You can tag a span
    continuedSpan.tag("taxValue", taxValue);
    // ...
    // You can log an event on a span
    continuedSpan.annotate("taxCalculated");
}
finally {
    // Once done remember to flush the span. That means that
    // it will get reported but the span itself is not yet finished
    continuedSpan.flush();
}
```

156.3. Creating a Span with an explicit Parent

You might want to start a new span and provide an explicit parent of that span. Assume that the parent of a span is in one thread and you want to start a new span in another thread. In Brave, whenever you call `nextSpan()`, it creates a span in reference to the span that is currently in scope. You can put the span in scope and then call `nextSpan()`, as shown in the following example:

```
// let's assume that we're in a thread Y and we've received
// the `initialSpan` from thread X. `initialSpan` will be the parent
// of the `newSpan`
Span newSpan = null;
try (Tracer.SpanInScope ws = this.tracer.withSpanInScope(initialSpan)) {
    newSpan = this.tracer.nextSpan().name("calculateCommission");
    // ...
    // You can tag a span
    newSpan.tag("commissionValue", commissionValue);
    // ...
    // You can log an event on a span
    newSpan.annotate("commissionCalculated");
}
finally {
    // Once done remember to finish the span. This will allow collecting
    // the span to send it to Zipkin. The tags and events set on the
    // newSpan will not be present on the parent
    if (newSpan != null) {
        newSpan.finish();
    }
}
```

After creating such a span, you must finish it. Otherwise it is not reported (for example, to Zipkin).

Chapter 157. Naming spans

Picking a span name is not a trivial task. A span name should depict an operation name. The name should be low cardinality, so it should not include identifiers.

Since there is a lot of instrumentation going on, some span names are artificial:

- `controller-method-name` when received by a Controller with a method name of `controllerMethodName`
- `async` for asynchronous operations done with wrapped `Callable` and `Runnable` interfaces.
- Methods annotated with `@Scheduled` return the simple name of the class.

Fortunately, for asynchronous processing, you can provide explicit naming.

157.1. `@SpanName` Annotation

You can name the span explicitly by using the `@SpanName` annotation, as shown in the following example:

```
@SpanName("calculateTax")
class TaxCountingRunnable implements Runnable {

    @Override
    public void run() {
        // perform logic
    }

}
```

In this case, when processed in the following manner, the span is named `calculateTax`:

```
Runnable runnable = new TraceRunnable(this.tracing, spanNamer,
        new TaxCountingRunnable());
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

157.2. `toString()` method

It is pretty rare to create separate classes for `Runnable` or `Callable`. Typically, one creates an anonymous instance of those classes. You cannot annotate such classes. To overcome that limitation, if there is no `@SpanName` annotation present, we check whether the class has a custom implementation of the `toString()` method.

Running such code leads to creating a span named `calculateTax`, as shown in the following example:

```java
Runnable runnable = new TraceRunnable(this.tracing, spanNamer, new Runnable() {
    @Override
    public void run() {
        // perform logic
    }

    @Override
    public String toString() {
        return "calculateTax";
    }
});
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

Chapter 158. Managing Spans with Annotations

You can manage spans with a variety of annotations.

158.1. Rationale

There are a number of good reasons to manage spans with annotations, including:

- API-agnostic means to collaborate with a span. Use of annotations lets users add to a span with no library dependency on a span api. Doing so lets Sleuth change its core API to create less impact to user code.
- Reduced surface area for basic span operations. Without this feature, you must use the span api, which has lifecycle commands that could be used incorrectly. By only exposing scope, tag, and log functionality, you can collaborate without accidentally breaking span lifecycle.
- Collaboration with runtime generated code. With libraries such as Spring Data and Feign, the implementations of interfaces are generated at runtime. Consequently, span wrapping of objects was tedious. Now you can provide annotations over interfaces and the arguments of those interfaces.

158.2. Creating New Spans

If you do not want to create local spans manually, you can use the `@NewSpan` annotation. Also, we provide the `@SpanTag` annotation to add tags in an automated fashion.

Now we can consider some examples of usage.

```
@NewSpan
void testMethod();
```

Annotating the method without any parameter leads to creating a new span whose name equals the annotated method name.

```
@NewSpan("customNameOnTestMethod4")
void testMethod4();
```

If you provide the value in the annotation (either directly or by setting the `name` parameter), the created span has the provided value as the name.

```
// method declaration
@NewSpan(name = "customNameOnTestMethod5")
void testMethod5(@SpanTag("testTag") String param);

// and method execution
this.testBean.testMethod5("test");
```

You can combine both the name and a tag. Let's focus on the latter. In this case, the value of the annotated method's parameter runtime value becomes the value of the tag. In our sample, the tag key is `testTag`, and the tag value is `test`.

```
@NewSpan(name = "customNameOnTestMethod3")
@Override
public void testMethod3() {
}
```

You can place the `@NewSpan` annotation on both the class and an interface. If you override the interface's method and provide a different value for the `@NewSpan` annotation, the most concrete one wins (in this case `customNameOnTestMethod3` is set).

158.3. Continuing Spans

If you want to add tags and annotations to an existing span, you can use the `@ContinueSpan` annotation, as shown in the following example:

```
// method declaration
@ContinueSpan(log = "testMethod11")
void testMethod11(@SpanTag("testTag11") String param);

// method execution
this.testBean.testMethod11("test");
this.testBean.testMethod13();
```

(Note that, in contrast with the `@NewSpan` annotation ,you can also add logs with the `log` parameter.)

That way, the span gets continued and:

- Log entries named `testMethod11.before` and `testMethod11.after` are created.
- If an exception is thrown, a log entry named `testMethod11.afterFailure` is also created.
- A tag with a key of `testTag11` and a value of `test` is created.

158.4. Advanced Tag Setting

There are 3 different ways to add tags to a span. All of them are controlled by the `SpanTag` annotation. The precedence is as follows:

1. Try with a bean of `TagValueResolver` type and a provided name.
2. If the bean name has not been provided, try to evaluate an expression. We search for a `TagValueExpressionResolver` bean. The default implementation uses SPEL expression resolution. **IMPORTANT** You can only reference properties from the SPEL expression. Method execution is not allowed due to security constraints.
3. If we do not find any expression to evaluate, return the `toString()` value of the parameter.

158.4.1. Custom extractor

The value of the tag for the following method is computed by an implementation of `TagValueResolver` interface. Its class name has to be passed as the value of the `resolver` attribute.

Consider the following annotated method:

```
@NewSpan
public void getAnnotationForTagValueResolver(
        @SpanTag(key = "test", resolver = TagValueResolver.class) String test) {
}
```

Now further consider the following `TagValueResolver` bean implementation:

```
@Bean(name = "myCustomTagValueResolver")
public TagValueResolver tagValueResolver() {
    return parameter -> "Value from myCustomTagValueResolver";
}
```

The two preceding examples lead to setting a tag value equal to `Value from myCustomTagValueResolver`.

158.4.2. Resolving Expressions for a Value

Consider the following annotated method:

```
@NewSpan
public void getAnnotationForTagValueExpression(@SpanTag(key = "test",
        expression = "'hello' + ' characters'") String test) {
}
```

No custom implementation of a `TagValueExpressionResolver` leads to evaluation of the SPEL expression, and a tag with a value of `4 characters` is set on the span. If you want to use some other expression resolution mechanism, you can create your own implementation of the bean.

158.4.3. Using the `toString()` method

Consider the following annotated method:

```
@NewSpan
public void getAnnotationForArgumentToString(@SpanTag("test") Long param) {
}
```

Running the preceding method with a value of `15` leads to setting a tag with a String value of `"15"`.

Chapter 159. Customizations

159.1. Customizers

With Brave 5.7 you have various options of providing customizers for your project. Brave ships with

- `TracingCustomizer` - allows configuration plugins to collaborate on building an instance of `Tracing`.
- `CurrentTraceContextCustomizer` - allows configuration plugins to collaborate on building an instance of `CurrentTraceContext`.
- `ExtraFieldCustomizer` - allows configuration plugins to collaborate on building an instance of `ExtraFieldPropagation.Factory`.

Sleuth will search for beans of those types and automatically apply customizations.

159.2. HTTP

159.2.1. Data Policy

The default span data policy for HTTP requests is described in Brave: github.com/openzipkin/brave/tree/master/instrumentation/http#span-data-policy

To add different data to the span, you need to register a bean of type `brave.http.HttpRequestParser` or `brave.http.HttpResponseParser` based on when the data is collected.

The bean names correspond to the request or response side, and whether it is a client or server. For example, `sleuthHttpClientRequestParser` changes what is collected before a client request is sent to the server.

For your convenience `@HttpClientRequestParser`, `@HttpClientResponseParser` and corresponding server annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Here’s an example adding the HTTP url in addition to defaults:

```
@Configuration
class Config {
  @Bean(name = { HttpClientRequestParser.NAME, HttpServerRequestParser.NAME })
  HttpRequestParser sleuthHttpServerRequestParser() {
      return (req, context, span) -> {
          HttpRequestParser.DEFAULT.parse(req, context, span);
          String url = req.url();
          if (url != null) {
              span.tag("http.url", url);
          }
      };
  }
}
```

159.2.2. Sampling

If client /server sampling is required, just register a bean of type `brave.sampler.SamplerFunction<HttpRequest>` and name the bean `sleuthHttpClientSampler` for client sampler and `sleuthHttpServerSampler` for server sampler.

For your convenience the `@HttpClientSampler` and `@HttpServerSampler` annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Check out Brave's code to see an example of how to make a path-based sampler github.com/openzipkin/brave/tree/master/instrumentation/http#sampling-policy

If you want to completely rewrite the `HttpTracing` bean you can use the `SkipPatternProvider` interface to retrieve the URL `Pattern` for spans that should be not sampled. Below you can see an example of usage of `SkipPatternProvider` inside a server side, `Sampler<HttpRequest>`.

```
@Configuration
class Config {
  @Bean(name = HttpServerSampler.NAME)
  SamplerFunction<HttpRequest> myHttpSampler(SkipPatternProvider provider) {
      Pattern pattern = provider.skipPattern();
      return request -> {
          String url = request.path();
          boolean shouldSkip = pattern.matcher(url).matches();
          if (shouldSkip) {
              return false;
          }
          return null;
      };
  }
}
```

159.3. TracingFilter

You can also modify the behavior of the `TracingFilter`, which is the component that is responsible for processing the input HTTP request and adding tags basing on the HTTP response. You can customize the tags or modify the response headers by registering your own instance of the `TracingFilter` bean.

In the following example, we register the `TracingFilter` bean, add the `ZIPKIN-TRACE-ID` response header containing the current Span's trace id, and add a tag with key `custom` and a value `tag` to the span.

```
@Component
@Order(TraceWebServletAutoConfiguration.TRACING_FILTER_ORDER + 1)
class MyFilter extends GenericFilterBean {

    private final Tracer tracer;

    MyFilter(Tracer tracer) {
        this.tracer = tracer;
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        Span currentSpan = this.tracer.currentSpan();
        if (currentSpan == null) {
            chain.doFilter(request, response);
            return;
        }
        // for readability we're returning trace id in a hex form
        ((HttpServletResponse) response).addHeader("ZIPKIN-TRACE-ID",
                currentSpan.context().traceIdString());
        // we can also add some custom tags
        currentSpan.tag("custom", "tag");
        chain.doFilter(request, response);
    }

}
```

159.4. Messaging

Sleuth automatically configures the `MessagingTracing` bean which serves as a foundation for Messaging instrumentation such as Kafka or JMS.

If a customization of producer / consumer sampling of messaging traces is required, just register a bean of type `brave.sampler.SamplerFunction<MessagingRequest>` and name the bean `sleuthProducerSampler` for producer sampler and `sleuthConsumerSampler` for consumer sampler.

For your convenience the `@ProducerSampler` and `@ConsumerSampler` annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Ex. Here's a sampler that traces 100 consumer requests per second, except for the "alerts" channel. Other requests will use a global rate provided by the `Tracing` component.

```
@Configuration
class Config {
}
```

For more, see github.com/openzipkin/brave/tree/master/instrumentation/messaging#sampling-policy

159.5. RPC

Sleuth automatically configures the `RpcTracing` bean which serves as a foundation for RPC instrumentation such as gRPC or Dubbo.

If a customization of client / server sampling of the RPC traces is required, just register a bean of type `brave.sampler.SamplerFunction<RpcRequest>` and name the bean `sleuthRpcClientSampler` for client sampler and `sleuthRpcServerSampler` for server sampler.

For your convenience the `@RpcClientSampler` and `@RpcServerSampler` annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Ex. Here's a sampler that traces 100 "GetUserToken" server requests per second. This doesn't start new traces for requests to the health check service. Other requests will use the global sampling configuration.

```
@Configuration
class Config {
  @Bean(name = RpcServerSampler.NAME)
  SamplerFunction<RpcRequest> myRpcSampler() {
      Matcher<RpcRequest> userAuth = and(serviceEquals("users.UserService"),
              methodEquals("GetUserToken"));
      return RpcRuleSampler.newBuilder()
              .putRule(serviceEquals("grpc.health.v1.Health"), Sampler.NEVER_SAMPLE)
              .putRule(userAuth, RateLimitingSampler.create(100)).build();
  }
}
```

For more, see github.com/openzipkin/brave/tree/master/instrumentation/rpc#sampling-policy

159.6. Custom service name

By default, Sleuth assumes that, when you send a span to Zipkin, you want the span's service name to be equal to the value of the `spring.application.name` property. That is not always the case, though.

There are situations in which you want to explicitly provide a different service name for all spans coming from your application. To achieve that, you can pass the following property to your application to override that value (the example is for a service named `myService`):

```
spring.zipkin.service.name: myService
```

159.7. Customization of Reported Spans

Before reporting spans (for example, to Zipkin) you may want to modify that span in some way. You can do so by implementing a `SpanHandler`.

In Sleuth, we generate spans with a fixed name. Some users want to modify the name depending on values of tags. You can implement the `SpanHandler` interface to alter that name.

The following example shows how to register two beans that implement `SpanHandler`:

```
@Bean
SpanHandler handlerOne() {
    return new SpanHandler() {
        @Override
        public boolean end(TraceContext traceContext, MutableSpan span,
                Cause cause) {
            span.name("foo");
            return true; // keep this span
        }
    };
}

@Bean
SpanHandler handlerTwo() {
    return new SpanHandler() {
        @Override
        public boolean end(TraceContext traceContext, MutableSpan span,
                Cause cause) {
            span.name(span.name() + " bar");
            return true; // keep this span
        }
    };
}
```

The preceding example results in changing the name of the reported span to `foo bar`, just before it gets reported (for example, to Zipkin).

159.8. Host Locator

This section is about defining **host** from service discovery. It is **NOT** about finding Zipkin through service discovery.

To define the host that corresponds to a particular span, we need to resolve the host name and port. The default approach is to take these values from server properties. If those are not set, we try to retrieve the host name from the network interfaces.

If you have the discovery client enabled and prefer to retrieve the host address from the registered instance in a service registry, you have to set the `spring.zipkin.locator.discovery.enabled` property (it is applicable for both HTTP-based and Stream-based span reporting), as follows:

```
spring.zipkin.locator.discovery.enabled: true
```

Chapter 160. Sending Spans to Zipkin

By default, if you add `spring-cloud-starter-zipkin` as a dependency to your project, when the span is closed, it is sent to Zipkin over HTTP. The communication is asynchronous. You can configure the URL by setting the `spring.zipkin.baseUrl` property, as follows:

```
spring.zipkin.baseUrl: https://192.168.99.100:9411/
```

If you want to find Zipkin through service discovery, you can pass the Zipkin's service ID inside the URL, as shown in the following example for `zipkinserver` service ID:

```
spring.zipkin.baseUrl: https://zipkinserver/
```

To disable this feature just set `spring.zipkin.discoveryClientEnabled` to `false.

When the Discovery Client feature is enabled, Sleuth uses `LoadBalancerClient` to find the URL of the Zipkin Server. It means that you can set up the load balancing configuration e.g. via Ribbon.

```
zipkinserver:
  ribbon:
    ListOfServers: host1,host2
```

If you have web, rabbit, activemq or kafka together on the classpath, you might need to pick the means by which you would like to send spans to zipkin. To do so, set `web`, `rabbit`, `activemq` or `kafka` to the `spring.zipkin.sender.type` property. The following example shows setting the sender type for `web`:

```
spring.zipkin.sender.type: web
```

To customize the `RestTemplate` that sends spans to Zipkin via HTTP, you can register the `ZipkinRestTemplateCustomizer` bean.

```
@Configuration
class MyConfig {
    @Bean ZipkinRestTemplateCustomizer myCustomizer() {
        return new ZipkinRestTemplateCustomizer() {
            @Override
            void customize(RestTemplate restTemplate) {
                // customize the RestTemplate
            }
        };
    }
}
```

If, however, you would like to control the full process of creating the `RestTemplate` object, you will have to create a bean of `zipkin2.reporter.Sender` type.

```
@Bean Sender myRestTemplateSender(ZipkinProperties zipkin,
        ZipkinRestTemplateCustomizer zipkinRestTemplateCustomizer) {
    RestTemplate restTemplate = mySuperCustomRestTemplate();
    zipkinRestTemplateCustomizer.customize(restTemplate);
    return myCustomSender(zipkin, restTemplate);
}
```

Chapter 161. Zipkin Stream Span Consumer

We recommend using Zipkin's native support for message-based span sending. Starting from the Edgware release, the Zipkin Stream server is deprecated. In the Finchley release, it got removed.

If for some reason you need to create the deprecated Stream Zipkin server, see the Dalston Documentation.

Chapter 162. Integrations

162.1. OpenTracing

Spring Cloud Sleuth is compatible with OpenTracing. If you have OpenTracing on the classpath, we automatically register the OpenTracing `Tracer` bean. If you wish to disable this, set `spring.sleuth.opentracing.enabled` to `false`

162.2. Runnable and Callable

If you wrap your logic in `Runnable` or `Callable`, you can wrap those classes in their Sleuth representative, as shown in the following example for `Runnable`:

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // do some work
    }

    @Override
    public String toString() {
        return "spanNameFromToStringMethod";
    }
};
// Manual `TraceRunnable` creation with explicit "calculateTax" Span name
Runnable traceRunnable = new TraceRunnable(this.tracing, spanNamer, runnable,
        "calculateTax");
// Wrapping `Runnable` with `Tracing`. That way the current span will be available
// in the thread of `Runnable`
Runnable traceRunnableFromTracer = this.tracing.currentTraceContext()
        .wrap(runnable);
```

The following example shows how to do so for `Callable`:

```java
Callable<String> callable = new Callable<String>() {
    @Override
    public String call() throws Exception {
        return someLogic();
    }

    @Override
    public String toString() {
        return "spanNameFromToStringMethod";
    }
};
// Manual `TraceCallable` creation with explicit "calculateTax" Span name
Callable<String> traceCallable = new TraceCallable<>(this.tracing, spanNamer,
        callable, "calculateTax");
// Wrapping `Callable` with `Tracing`. That way the current span will be available
// in the thread of `Callable`
Callable<String> traceCallableFromTracer = this.tracing.currentTraceContext()
        .wrap(callable);
```

That way, you ensure that a new span is created and closed for each execution.

162.3. Spring Cloud CircuitBreaker

If you have Spring Cloud CircuitBreaker on the classpath, we will wrap the passed command `Supplier` and the fallback `Function` in its trace representations. In order to disable this instrumentation set `spring.sleuth.circuitbreaker.enabled` to `false`.

162.4. Hystrix

162.4.1. Custom Concurrency Strategy

We register a custom `HystrixConcurrencyStrategy` called `TraceCallable` that wraps all `Callable` instances in their Sleuth representative. The strategy either starts or continues a span, depending on whether tracing was already going on before the Hystrix command was called. Optionally, you can set `spring.sleuth.hystrix.strategy.passthrough` to `true` to just propagate the trace context to the Hystrix execution thread if you don't wish to start a new span. To disable the custom Hystrix Concurrency Strategy, set the `spring.sleuth.hystrix.strategy.enabled` to `false`.

162.4.2. Manual Command setting

Assume that you have the following `HystrixCommand`:

```
HystrixCommand<String> hystrixCommand = new HystrixCommand<String>(setter) {
    @Override
    protected String run() throws Exception {
        return someLogic();
    }
};
```

To pass the tracing information, you have to wrap the same logic in the Sleuth version of the `HystrixCommand`, which is called `TraceCommand`, as shown in the following example:

```
TraceCommand<String> traceCommand = new TraceCommand<String>(tracer, setter) {
    @Override
    public String doRun() throws Exception {
        return someLogic();
    }
};
```

162.5. RxJava

We registering a custom `RxJavaSchedulersHook` that wraps all `Action0` instances in their Sleuth representative, which is called `TraceAction`. The hook either starts or continues a span, depending on whether tracing was already going on before the Action was scheduled. To disable the custom `RxJavaSchedulersHook`, set the `spring.sleuth.rxjava.schedulers.hook.enabled` to `false`.

You can define a list of regular expressions for thread names for which you do not want spans to be created. To do so, provide a comma-separated list of regular expressions in the `spring.sleuth.rxjava.schedulers.ignoredthreads` property.

The suggest approach to reactive programming and Sleuth is to use the Reactor support.

162.6. HTTP integration

Features from this section can be disabled by setting the `spring.sleuth.web.enabled` property with value equal to `false`.

162.6.1. HTTP Filter

Through the `TracingFilter`, all sampled incoming requests result in creation of a Span. That Span's name is `http:` + the path to which the request was sent. For example, if the request was sent to `/this/that` then the name will be `http:/this/that`. You can configure which URIs you would like to skip by setting the `spring.sleuth.web.skipPattern` property. If you have `ManagementServerProperties` on classpath, its value of `contextPath` gets appended to the provided skip pattern. If you want to reuse the Sleuth's default skip patterns and just append your own, pass those patterns by using the `spring.sleuth.web.additionalSkipPattern`.

By default, all the spring boot actuator endpoints are automatically added to the skip pattern. If you want to disable this behaviour set `spring.sleuth.web.ignore-auto-configured-skip-patterns` to `true`.

To change the order of tracing filter registration, please set the `spring.sleuth.web.filter-order` property.

To disable the filter that logs uncaught exceptions you can disable the `spring.sleuth.web.exception-throwing-filter-enabled` property.

162.6.2. HandlerInterceptor

Since we want the span names to be precise, we use a `TraceHandlerInterceptor` that either wraps an existing `HandlerInterceptor` or is added directly to the list of existing `HandlerInterceptors`. The `TraceHandlerInterceptor` adds a special request attribute to the given `HttpServletRequest`. If the the `TracingFilter` does not see this attribute, it creates a “fallback” span, which is an additional span created on the server side so that the trace is presented properly in the UI. If that happens, there is probably missing instrumentation. In that case, please file an issue in Spring Cloud Sleuth.

162.6.3. Async Servlet support

If your controller returns a `Callable` or a `WebAsyncTask`, Spring Cloud Sleuth continues the existing span instead of creating a new one.

162.6.4. WebFlux support

Through `TraceWebFilter`, all sampled incoming requests result in creation of a Span. That Span’s name is `http:` + the path to which the request was sent. For example, if the request was sent to `/this/that`, the name is `http:/this/that`. You can configure which URIs you would like to skip by using the `spring.sleuth.web.skipPattern` property. If you have `ManagementServerProperties` on the classpath, its value of `contextPath` gets appended to the provided skip pattern. If you want to reuse Sleuth’s default skip patterns and append your own, pass those patterns by using the `spring.sleuth.web.additionalSkipPattern`.

To change the order of tracing filter registration, please set the `spring.sleuth.web.filter-order` property.

162.6.5. Dubbo RPC support

Via the integration with Brave, Spring Cloud Sleuth supports Dubbo. It’s enough to add the `brave-instrumentation-dubbo` dependency:

```
<dependency>
    <groupId>io.zipkin.brave</groupId>
    <artifactId>brave-instrumentation-dubbo</artifactId>
</dependency>
```

You need to also set a `dubbo.properties` file with the following contents:

```
dubbo.provider.filter=tracing
dubbo.consumer.filter=tracing
```

You can read more about Brave - Dubbo integration here. An example of Spring Cloud Sleuth and Dubbo can be found here.

162.7. HTTP Client Integration

162.7.1. Synchronous Rest Template

We inject a `RestTemplate` interceptor to ensure that all the tracing information is passed to the requests. Each time a call is made, a new Span is created. It gets closed upon receiving the response. To block the synchronous `RestTemplate` features, set `spring.sleuth.web.client.enabled` to `false`.

You have to register `RestTemplate` as a bean so that the interceptors get injected. If you create a `RestTemplate` instance with a `new` keyword, the instrumentation does NOT work.

162.7.2. Asynchronous Rest Template

Starting with Sleuth `2.0.0`, we no longer register a bean of `AsyncRestTemplate` type. It is up to you to create such a bean. Then we instrument it.

To block the `AsyncRestTemplate` features, set `spring.sleuth.web.async.client.enabled` to `false`. To disable creation of the default `TraceAsyncClientHttpRequestFactoryWrapper`, set `spring.sleuth.web.async.client.factory.enabled` to `false`. If you do not want to create `AsyncRestClient` at all, set `spring.sleuth.web.async.client.template.enabled` to `false`.

Multiple Asynchronous Rest Templates

Sometimes you need to use multiple implementations of the Asynchronous Rest Template. In the following snippet, you can see an example of how to set up such a custom `AsyncRestTemplate`:

```
@Configuration
@EnableAutoConfiguration
static class Config {

    @Bean(name = "customAsyncRestTemplate")
    public AsyncRestTemplate traceAsyncRestTemplate() {
        return new AsyncRestTemplate(asyncClientFactory(),
                clientHttpRequestFactory());
    }

    private ClientHttpRequestFactory clientHttpRequestFactory() {
        ClientHttpRequestFactory clientHttpRequestFactory = new
CustomClientHttpRequestFactory();
        // CUSTOMIZE HERE
        return clientHttpRequestFactory;
    }

    private AsyncClientHttpRequestFactory asyncClientFactory() {
        AsyncClientHttpRequestFactory factory = new
CustomAsyncClientHttpRequestFactory();
        // CUSTOMIZE HERE
        return factory;
    }

}
```

162.7.3. `WebClient`

We inject a `ExchangeFilterFunction` implementation that creates a span and, through on-success and on-error callbacks, takes care of closing client-side spans.

To block this feature, set `spring.sleuth.web.client.enabled` to `false`.

You have to register either a `WebClient` or `WebClient.Builder` as a bean so that the tracing instrumentation gets applied. If you manually create a `WebClient` or `WebClient.Builder`, the instrumentation does NOT work.

162.7.4. Traverson

If you use the Traverson library, you can inject a `RestTemplate` as a bean into your Traverson object. Since `RestTemplate` is already intercepted, you get full support for tracing in your client. The following pseudo code shows how to do that:

```
@Autowired RestTemplate restTemplate;

Traverson traverson = new Traverson(URI.create("https://some/address"),
    MediaType.APPLICATION_JSON,
MediaType.APPLICATION_JSON_UTF8).setRestOperations(restTemplate);
// use Traverson
```

162.7.5. Apache `HttpClientBuilder` and `HttpAsyncClientBuilder`

We instrument the `HttpClientBuilder` and `HttpAsyncClientBuilder` so that tracing context gets injected to the sent requests.

To block these features, set `spring.sleuth.web.client.enabled` to `false`.

162.7.6. Netty `HttpClient`

We instrument the Netty's `HttpClient`.

To block this feature, set `spring.sleuth.web.client.enabled` to `false`.

You have to register `HttpClient` as a bean so that the instrumentation happens. If you create a `HttpClient` instance with a `new` keyword, the instrumentation does NOT work.

162.7.7. `UserInfoRestTemplateCustomizer`

We instrument the Spring Security's `UserInfoRestTemplateCustomizer`.

To block this feature, set `spring.sleuth.web.client.enabled` to `false`.

162.8. Feign

By default, Spring Cloud Sleuth provides integration with Feign through `TraceFeignClientAutoConfiguration`. You can disable it entirely by setting `spring.sleuth.feign.enabled` to `false`. If you do so, no Feign-related instrumentation take place.

Part of Feign instrumentation is done through a `FeignBeanPostProcessor`. You can disable it by setting `spring.sleuth.feign.processor.enabled` to `false`. If you set it to `false`, Spring Cloud Sleuth does not instrument any of your custom Feign components. However, all the default instrumentation is still there.

162.9. gRPC

Spring Cloud Sleuth provides instrumentation for gRPC through `TraceGrpcAutoConfiguration`. You can disable it entirely by setting `spring.sleuth.grpc.enabled` to `false`.

162.9.1. Variant 1

Dependencies

The gRPC integration relies on two external libraries to instrument clients and servers and both of those libraries must be on the class path to enable the instrumentation.

Maven:

```
<dependency>
    <groupId>io.github.lognet</groupId>
    <artifactId>grpc-spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.brave</groupId>
    <artifactId>brave-instrumentation-grpc</artifactId>
</dependency>
```

Gradle:

```
compile("io.github.lognet:grpc-spring-boot-starter")
compile("io.zipkin.brave:brave-instrumentation-grpc")
```

Server Instrumentation

Spring Cloud Sleuth leverages grpc-spring-boot-starter to register Brave's gRPC server interceptor with all services annotated with `@GRpcService`.

Client Instrumentation

gRPC clients leverage a `ManagedChannelBuilder` to construct a `ManagedChannel` used to communicate to the gRPC server. The native `ManagedChannelBuilder` provides static methods as entry points for construction of `ManagedChannel` instances, however, this mechanism is outside the influence of the Spring application context.

Spring Cloud Sleuth provides a `SpringAwareManagedChannelBuilder` that can be customized through the Spring application context and injected by gRPC clients. **This builder must be used when creating `ManagedChannel` instances.**

Sleuth creates a `TracingManagedChannelBuilderCustomizer` which inject Brave's client interceptor into the `SpringAwareManagedChannelBuilder`.

162.9.2. Variant 2

Grpc Spring Boot Starter automatically detects the presence of Spring Cloud Sleuth and brave's instrumentation for gRPC and registers the necessary client and/or server tooling.

162.10. Asynchronous Communication

162.10.1. `@Async` Annotated methods

In Spring Cloud Sleuth, we instrument async-related components so that the tracing information is passed between threads. You can disable this behavior by setting the value of `spring.sleuth.async.enabled` to `false`.

If you annotate your method with `@Async`, we automatically create a new Span with the following characteristics:

- If the method is annotated with `@SpanName`, the value of the annotation is the Span's name.
- If the method is not annotated with `@SpanName`, the Span name is the annotated method name.
- The span is tagged with the method's class name and method name.

162.10.2. `@Scheduled` Annotated Methods

In Spring Cloud Sleuth, we instrument scheduled method execution so that the tracing information is passed between threads. You can disable this behavior by setting the value of `spring.sleuth.scheduled.enabled` to `false`.

If you annotate your method with `@Scheduled`, we automatically create a new span with the following characteristics:

- The span name is the annotated method name.
- The span is tagged with the method's class name and method name.

If you want to skip span creation for some `@Scheduled` annotated classes, you can set the `spring.sleuth.scheduled.skipPattern` with a regular expression that matches the fully qualified name of the `@Scheduled` annotated class. If you use `spring-cloud-sleuth-stream` and `spring-cloud-netflix-hystrix-stream` together, a span is created for each Hystrix metrics and sent to Zipkin. This behavior may be annoying. That's why, by default, `spring.sleuth.scheduled.skipPattern=org.springframework.cloud.netflix.hystrix.stream.HystrixStreamTask`.

162.10.3. Executor, ExecutorService, and ScheduledExecutorService

We provide `LazyTraceExecutor`, `TraceableExecutorService`, and `TraceableScheduledExecutorService`. Those implementations create spans each time a new task is submitted, invoked, or scheduled.

The following example shows how to pass tracing information with `TraceableExecutorService` when working with `CompletableFuture`:

```
CompletableFuture<Long> completableFuture = CompletableFuture.supplyAsync(() -> {
    // perform some logic
    return 1_000_000L;
}, new TraceableExecutorService(beanFactory, executorService,
        // 'calculateTax' explicitly names the span - this param is optional
        "calculateTax"));
```

Sleuth does not work with `parallelStream()` out of the box. If you want to have the tracing information propagated through the stream, you have to use the approach with `supplyAsync(…)`, as shown earlier.

If there are beans that implement the `Executor` interface that you would like to exclude from span creation, you can use the `spring.sleuth.async.ignored-beans` property where you can provide a list of bean names.

Customization of Executors

Sometimes, you need to set up a custom instance of the `AsyncExecutor`. The following example shows how to set up such a custom `Executor`:

```
@Configuration
@EnableAutoConfiguration
@EnableAsync
// add the infrastructure role to ensure that the bean gets auto-proxied
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
static class CustomExecutorConfig extends AsyncConfigurerSupport {

    @Autowired
    BeanFactory beanFactory;

    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        // CUSTOMIZE HERE
        executor.setCorePoolSize(7);
        executor.setMaxPoolSize(42);
        executor.setQueueCapacity(11);
        executor.setThreadNamePrefix("MyExecutor-");
        // DON'T FORGET TO INITIALIZE
        executor.initialize();
        return new LazyTraceExecutor(this.beanFactory, executor);
    }

}
```

To ensure that your configuration gets post processed, remember to add the `@Role(BeanDefinition.ROLE_INFRASTRUCTURE)` on your `@Configuration` class

162.11. Messaging

Features from this section can be disabled by setting the `spring.sleuth.messaging.enabled` property with value equal to `false`.

162.11.1. Spring Integration and Spring Cloud Stream

Spring Cloud Sleuth integrates with Spring Integration. It creates spans for publish and subscribe events. To disable Spring Integration instrumentation, set `spring.sleuth.integration.enabled` to `false`.

You can provide the `spring.sleuth.integration.patterns` pattern to explicitly provide the names of channels that you want to include for tracing. By default, all channels but `hystrixStreamOutput` channel are included.

When using the `Executor` to build a Spring Integration `IntegrationFlow`, you must use the untraced version of the `Executor`. Decorating the Spring Integration Executor Channel with `TraceableExecutorService` causes the spans to be improperly closed.

If you want to customize the way tracing context is read from and written to message headers, it's enough for you to register beans of types:

- `Propagation.Setter<MessageHeaderAccessor, String>` - for writing headers to the message
- `Propagation.Getter<MessageHeaderAccessor, String>` - for reading headers from the message

162.11.2. Spring RabbitMq

We instrument the `RabbitTemplate` so that tracing headers get injected into the message.

To block this feature, set `spring.sleuth.messaging.rabbit.enabled` to `false`.

162.11.3. Spring Kafka

We instrument the Spring Kafka's `ProducerFactory` and `ConsumerFactory` so that tracing headers get injected into the created Spring Kafka's `Producer` and `Consumer`.

To block this feature, set `spring.sleuth.messaging.kafka.enabled` to `false`.

162.11.4. Spring Kafka Streams

We instrument the `KafkaStreams KafkaClientSupplier` so that tracing headers get injected into the `Producer` and ``Consumer`s. A `KafkaStreamsTracing`` bean allows for further instrumentation through additional `TransformerSupplier` and `ProcessorSupplier` methods.

To block this feature, set `spring.sleuth.messaging.kafka.streams.enabled` to `false`.

162.11.5. Spring JMS

We instrument the `JmsTemplate` so that tracing headers get injected into the message. We also support `@JmsListener` annotated methods on the consumer side.

To block this feature, set `spring.sleuth.messaging.jms.enabled` to `false`.

We don't support baggage propagation for JMS

162.11.6. Spring Cloud AWS Messaging SQS

We instrument `@SqsListener` which is provided by `org.springframework.cloud:spring-cloud-aws-messaging` so that tracing headers get extracted from the message and a trace gets put into the context.

To block this feature, set `spring.sleuth.messaging.sqs.enabled` to `false`.

162.12. Zuul

We instrument the Zuul Ribbon integration by enriching the Ribbon requests with tracing information. To disable Zuul support, set the `spring.sleuth.zuul.enabled` property to `false`.

162.13. Redis

We set `tracing` property to Lettcue `ClientResources` instance to enable Brave tracing built in Lettuce . To disable Redis support, set the `spring.sleuth.redis.enabled` property to `false`.

162.14. Quartz

We instrument quartz jobs by adding Job/Trigger listeners to the Quartz Scheduler.

To turn off this feature, set the `spring.sleuth.quartz.enabled` property to `false`.

162.15. Project Reactor

For projects depending on Project Reactor such as Spring Cloud Gateway, we suggest turning the `spring.sleuth.reactor.decorate-on-each` option to `false`. That way an increased performance gain should be observed in comparison to the standard instrumentation mechanism. What this option does is it will wrap decorate `onLast` operator instead of `onEach` which will result in creation of far fewer objects. The downside of this is that when Project Reactor will change threads, the trace propagation will continue without issues, however anything relying on the `ThreadLocal` such as e.g. MDC entries can be buggy.

Chapter 163. Configuration properties

To see the list of all Sleuth related configuration properties please check the Appendix page.

Chapter 164. Running examples

You can see the running examples deployed in the Pivotal Web Services. Check them out at the following links:

- Zipkin for apps presented in the samples to the top. First make a request to Service 1 and then check out the trace in Zipkin.
- Zipkin for Brewery on PWS, its Github Code. Ensure that you've picked the lookback period of 7 days. If there are no traces, go to Presenting application and order some beers. Then check Zipkin for traces.

Spring Cloud Task Reference Guide

Michael Minella, Glenn Renfro, Jay Bryant :doctype: book :toc: :toclevels: 4 :source-highlighter: prettify :numbered: :icons: font :hide-uri-scheme: :spring-cloud-task-repo: snapshot :github-tag: master :spring-cloud-task-docs-version: current :spring-cloud-task-docs: docs.spring.io/spring-cloud-task/docs/{version}/reference :spring-cloud-task-docs-current: docs.spring.io/spring-cloud-task/docs/current-SNAPSHOT/reference/html/ :github-repo: spring-cloud/spring-cloud-task :github-raw: raw.github.com/spring-cloud/spring-cloud-netflix/master :github-code: github.com/spring-cloud/spring-cloud-netflix/tree/master :github-wiki: github.com/spring-cloud/spring-cloud-netflix/wiki :github-master-code: github.com/spring-cloud/spring-cloud-netflix/tree/master :sc-ext: java :sc-spring-boot: github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-boot/src/main/java/org/springframework/boot :dc-ext: html :dc-root: docs.spring.io/spring-cloud-task/docs/{spring-cloud-dataflow-docs-version}/api :dc-spring-boot: {dc-root}/org/springframework/boot :dependency-management-plugin: github.com/spring-gradle-plugins/dependency-management-plugin :dependency-management-plugin-documentation: {dependency-management-plugin}/blob/master/README.md :spring-boot-maven-plugin-site: docs.spring.io/spring-boot/docs/{spring-boot-docs-version}/maven-plugin :spring-reference: docs.spring.io/spring/docs/{spring-docs-version}/spring-framework-reference/htmlsingle :spring-security-reference: docs.spring.io/spring-security/site/docs/{spring-security-docs-version}/reference/htmlsingle :spring-javadoc: docs.spring.io/spring/docs/{spring-docs-version}/javadoc-api/org/springframework :spring-amqp-javadoc: docs.spring.io/spring-amqp/docs/current/api/org/springframework/amqp :spring-data-javadoc: docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa :spring-data-commons-javadoc: docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data :spring-data-mongo-javadoc: docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb :spring-data-rest-javadoc: docs.spring.io/spring-data/rest/docs/current/api/org/springframework/data/rest :gradle-userguide: www.gradle.org/docs/current/userguide :propdeps-plugin: github.com/spring-projects/gradle-plugins/tree/master/propdeps-plugin :ant-manual: ant.apache.org/manual :attributes: allow-uri-read

© 2009-2018 Pivotal Software, Inc. All rights reserved.

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.

Preface

This section provides a brief overview of the Spring Cloud Task reference documentation. Think of it as a map for the rest of the document. You can read this reference guide in a linear fashion or you can skip sections if something does not interest you.

Chapter 165. About the documentation

The Spring Cloud Task reference guide is available in {spring-cloud-task-docs}/html[html], {spring-cloud-task-docs}/pdf/spring-cloud-task-reference.pdf[pdf] and {spring-cloud-task-docs}/epub/spring-cloud-task-reference.epub[epub] formats. The latest copy is available at {spring-cloud-task-docs-current}.

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 166. Getting help

Having trouble with Spring Cloud Task? We would like to help!

- Ask a question. We monitor stackoverflow.com for questions tagged with `spring-cloud-task`.
- Report bugs with Spring Cloud Task at github.com/spring-cloud/spring-cloud-task/issues.

All of Spring Cloud Task is open source, including the documentation. If you find a problem with the docs or if you just want to improve them, please get involved.

Chapter 167. First Steps

If you are just getting started with Spring Cloud Task or with 'Spring' in general, we suggesting reading the getting-started.pdf chapter.

To get started from scratch, read the following sections: * “getting-started.pdf” * “getting-started.pdf” To follow the tutorial, read “getting-started.pdf” To run your example, read “getting-started.pdf”

Getting started

If you are just getting started with Spring Cloud Task, you should read this section. Here, we answer the basic “what?”, “how?”, and “why?” questions. We start with a gentle introduction to Spring Cloud Task. We then build a Spring Cloud Task application, discussing some core principles as we go.

Chapter 168. Introducing Spring Cloud Task

Spring Cloud Task makes it easy to create short-lived microservices. It provides capabilities that let short lived JVM processes be executed on demand in a production environment.

Chapter 169. System Requirements

You need to have Java installed (Java 8 or better). To build, you need to have Maven installed as well.

169.1. Database Requirements

Spring Cloud Task uses a relational database to store the results of an executed task. While you can begin developing a task without a database (the status of the task is logged as part of the task repository's updates), for production environments, you want to use a supported database. Spring Cloud Task currently supports the following databases:

- DB2
- H2
- HSQLDB
- MySql
- Oracle
- Postgres
- SqlServer

Chapter 170. Developing Your First Spring Cloud Task Application

A good place to start is with a simple "Hello, World!" application, so we create the Spring Cloud Task equivalent to highlight the features of the framework. Most IDEs have good support for Apache Maven, so we use it as the build tool for this project.

The 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 following steps by going to the Spring Initializr and creating a new project. Doing so automatically generates a new project structure so that you can start coding right away. We recommend experimenting with the Spring Initializr to become familiar with it.

170.1. Creating the Spring Task Project using Spring Initializr

Now we can create and test an application that prints `Hello, World!` to the console.

To do so:

1. Visit the Spring Initialzr site.
   a. Create a new Maven project with a **Group** name of `io.spring.demo` and an **Artifact** name of `helloworld`.
   b. In the Dependencies text box, type `task` and then select the `Cloud Task` dependency.
   c. In the Dependencies text box, type `jdbc` and then select the `JDBC` dependency.
   d. In the Dependencies text box, type `h2` and then select the `H2`. (or your favorite database)
   e. Click the **Generate Project** button
2. Unzip the helloworld.zip file and import the project into your favorite IDE.

170.2. Writing the Code

To finish our application, we need to update the generated `HelloworldApplication` with the following contents so that it launches a Task.

```java
package io.spring.demo.helloworld;

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

@SpringBootApplication
public class HelloworldApplication {

    @Bean
    public CommandLineRunner commandLineRunner() {
        return new HelloWorldCommandLineRunner();
    }

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

    public static class HelloWorldCommandLineRunner implements CommandLineRunner {

        @Override
        public void run(String... strings) throws Exception {
            System.out.println("Hello, World!");
        }
    }
}
```

While it may seem small, quite a bit is going on. For more about Spring Boot specifics, see the Spring Boot reference documentation.

Now we can open the `application.properties` file in `src/main/resources`. We need to configure two properties in `application.properties`:

- `application.name`: To set the application name (which is translated to the task name)
- `logging.level`: To set the logging for Spring Cloud Task to `DEBUG` in order to get a view of what is going on.

The following example shows how to do both:

```
logging.level.org.springframework.cloud.task=DEBUG
spring.application.name=helloWorld
```

170.2.1. Task Auto Configuration

When including Spring Cloud Task Starter dependency, Task auto configures all beans to bootstrap it's functionality. Part of this configuration registers the `TaskRepository` and the infrastructure for its use.

In our demo, the `TaskRepository` uses an embedded H2 database to record the results of a task. This H2 embedded database is not a practical solution for a production environment, since the H2 DB goes away once the task ends. However, for a quick getting-started experience, we can use this in our example as well as echoing to the logs what is being updated in that repository. In the Configuration section (later in this documentation), we cover how to customize the configuration of the pieces provided by Spring Cloud Task.

When our sample application runs, Spring Boot launches our `HelloWorldCommandLineRunner` and outputs our “Hello, World!” message to standard out. The `TaskLifecycleListener` records the start of the task and the end of the task in the repository.

170.2.2. The main method

The main method serves as the entry point to any java application. Our main method delegates to Spring Boot’s SpringApplication class.

170.2.3. The CommandLineRunner

Spring includes many ways to bootstrap an application’s logic. Spring Boot provides a convenient method of doing so in an organized manner through its `*Runner` interfaces (`CommandLineRunner` or `ApplicationRunner`). A well behaved task can bootstrap any logic by using one of these two runners.

The lifecycle of a task is considered from before the `*Runner#run` methods are executed to once they are all complete. Spring Boot lets an application use multiple `*Runner` implementations, as does Spring Cloud Task.

Any processing bootstrapped from mechanisms other than a `CommandLineRunner` or `ApplicationRunner` (by using `InitializingBean#afterPropertiesSet` for example) is not recorded by Spring Cloud Task.

170.3. Running the Example

At this point, our application should work. Since this application is Spring Boot-based, we can run it from the command line by using `$ mvn spring-boot:run` from the root of our application, as shown (with its output) in the following example:

```
$ mvn clean spring-boot:run
....... . . .
....... . . . (Maven log output here)
....... . . .


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.3.RELEASE)
```

```
2018-07-23 17:44:34.426  INFO 1978 --- [           main]
i.s.d.helloworld.HelloworldApplication   : Starting HelloworldApplication on Glenns-
MBP-2.attlocal.net with PID 1978 (/Users/glennrenfro/project/helloworld/target/classes
started by glennrenfro in /Users/glennrenfro/project/helloworld)
2018-07-23 17:44:34.430  INFO 1978 --- [           main]
i.s.d.helloworld.HelloworldApplication   : No active profile set, falling back to
default profiles: default
2018-07-23 17:44:34.472  INFO 1978 --- [           main]
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@1d24f32d:
startup date [Mon Jul 23 17:44:34 EDT 2018]; root of context hierarchy
2018-07-23 17:44:35.280  INFO 1978 --- [           main]
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2018-07-23 17:44:35.410  INFO 1978 --- [           main]
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2018-07-23 17:44:35.419 DEBUG 1978 --- [           main]
o.s.c.t.c.SimpleTaskConfiguration        : Using
org.springframework.cloud.task.configuration.DefaultTaskConfigurer TaskConfigurer
2018-07-23 17:44:35.420 DEBUG 1978 --- [           main]
o.s.c.t.c.DefaultTaskConfigurer          : No EntityManager was found, using
DataSourceTransactionManager
2018-07-23 17:44:35.522 DEBUG 1978 --- [           main]
o.s.c.t.r.s.TaskRepositoryInitializer    : Initializing task schema for h2 database
2018-07-23 17:44:35.525  INFO 1978 --- [           main]
o.s.jdbc.datasource.init.ScriptUtils     : Executing SQL script from class path
resource [org/springframework/cloud/task/schema-h2.sql]
2018-07-23 17:44:35.558  INFO 1978 --- [           main]
o.s.jdbc.datasource.init.ScriptUtils     : Executed SQL script from class path
resource [org/springframework/cloud/task/schema-h2.sql] in 33 ms.
2018-07-23 17:44:35.728  INFO 1978 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on
startup
2018-07-23 17:44:35.730  INFO 1978 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter        : Bean with name 'dataSource' has been
autodetected for JMX exposure
2018-07-23 17:44:35.733  INFO 1978 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter        : Located MBean 'dataSource': registering
with JMX server as MBean [com.zaxxer.hikari:name=dataSource,type=HikariDataSource]
2018-07-23 17:44:35.738  INFO 1978 --- [           main]
o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase 0
2018-07-23 17:44:35.762 DEBUG 1978 --- [           main]
o.s.c.t.r.support.SimpleTaskRepository   : Creating: TaskExecution{executionId=0,
parentExecutionId=null, exitCode=null, taskName='application', startTime=Mon Jul 23
17:44:35 EDT 2018, endTime=null, exitMessage='null', externalExecutionId='null',
errorMessage='null', arguments=[]}
2018-07-23 17:44:35.772  INFO 1978 --- [           main]
i.s.d.helloworld.HelloworldApplication   : Started HelloworldApplication in 1.625
seconds (JVM running for 4.764)
Hello, World!
2018-07-23 17:44:35.782 DEBUG 1978 --- [           main]
```

```
o.s.c.t.r.support.SimpleTaskRepository   : Updating: TaskExecution with executionId=1
with the following {exitCode=0, endTime=Mon Jul 23 17:44:35 EDT 2018,
exitMessage='null', errorMessage='null'}
```

The preceding output has three lines that of interest to us here:

- `SimpleTaskRepository` logged the creation of the entry in the `TaskRepository`.
- The execution of our `CommandLineRunner`, demonstrated by the “Hello, World!” output.
- `SimpleTaskRepository` logs the completion of the task in the `TaskRepository`.

A simple task application can be found in the samples module of the Spring Cloud Task Project here.

Features

This section goes into more detail about Spring Cloud Task, including how to use it, how to configure it, and the appropriate extension points.

Chapter 171. The lifecycle of a Spring Cloud Task

In most cases, the modern cloud environment is designed around the execution of processes that are not expected to end. If they do end, they are typically restarted. While most platforms do have some way to run a process that is not restarted when it ends, the results of that run are typically not maintained in a consumable way. Spring Cloud Task offers the ability to execute short-lived processes in an environment and record the results. Doing so allows for a microservices architecture around short-lived processes as well as longer running services through the integration of tasks by messages.

While this functionality is useful in a cloud environment, the same issues can arise in a traditional deployment model as well. When running Spring Boot applications with a scheduler such as cron, it can be useful to be able to monitor the results of the application after its completion.

Spring Cloud Task takes the approach that a Spring Boot application can have a start and an end and still be successful. Batch applications are one example of how processes that are expected to end (and that are often short-lived) can be helpful.

Spring Cloud Task records the lifecycle events of a given task. Most long-running processes, typified by most web applications, do not save their lifecycle events. The tasks at the heart of Spring Cloud Task do.

The lifecycle consists of a single task execution. This is a physical execution of a Spring Boot application configured to be a task (that is, it has the Sprint Cloud Task dependencies).

At the beginning of a task, before any `CommandLineRunner` or `ApplicationRunner` implementations have been run, an entry in the `TaskRepository` that records the start event is created. This event is triggered through `SmartLifecycle#start` being triggered by the Spring Framework. This indicates to the system that all beans are ready for use and comes before running any of the `CommandLineRunner` or `ApplicationRunner` implementations provided by Spring Boot.

The recording of a task only occurs upon the successful bootstrapping of an `ApplicationContext`. If the context fails to bootstrap at all, the task's run is not recorded.

Upon completion of all of the `*Runner#run` calls from Spring Boot or the failure of an `ApplicationContext` (indicated by an `ApplicationFailedEvent`), the task execution is updated in the repository with the results.

If the application requires the `ApplicationContext` to be closed at the completion of a task (all `*Runner#run` methods have been called and the task repository has been updated), set the property `spring.cloud.task.closecontextEnabled` to true.

171.1. The TaskExecution

The information stored in the `TaskRepository` is modeled in the `TaskExecution` class and consists of the following information:

Field	Description
`executionid`	The unique ID for the task's run.
`exitCode`	The exit code generated from an `ExitCodeExceptionMapper` implementation. If there is no exit code generated but an `ApplicationFailedEvent` is thrown, 1 is set. Otherwise, it is assumed to be 0.
`taskName`	The name for the task, as determined by the configured `TaskNameResolver`.
`startTime`	The time the task was started, as indicated by the `SmartLifecycle#start` call.
`endTime`	The time the task was completed, as indicated by the `ApplicationReadyEvent`.
`exitMessage`	Any information available at the time of exit. This can programmatically be set by a `TaskExecutionListener`.
`errorMessage`	If an exception is the cause of the end of the task (as indicated by an `ApplicationFailedEvent`), the stack trace for that exception is stored here.
`arguments`	A `List` of the string command line arguments as they were passed into the executable boot application.

171.2. Mapping Exit Codes

When a task completes, it tries to return an exit code to the OS. If we take a look at our original example, we can see that we are not controlling that aspect of our application. So, if an exception is thrown, the JVM returns a code that may or may not be of any use to you in debugging.

Consequently, Spring Boot provides an interface, `ExitCodeExceptionMapper`, that lets you map uncaught exceptions to exit codes. Doing so lets you indicate, at the level of exit codes, what went wrong. Also, by mapping exit codes in this manner, Spring Cloud Task records the returned exit code.

If the task terminates with a SIG-INT or a SIG-TERM, the exit code is zero unless otherwise specified within the code.

While the task is running, the exit code is stored as a null in the repository. Once the task completes, the appropriate exit code is stored based on the guidelines described earlier in this section.

Chapter 172. Configuration

Spring Cloud Task provides a ready-to-use configuration, as defined in the `DefaultTaskConfigurer` and `SimpleTaskConfiguration` classes. This section walks through the defaults and how to customize Spring Cloud Task for your needs.

172.1. DataSource

Spring Cloud Task uses a datasource for storing the results of task executions. By default, we provide an in-memory instance of H2 to provide a simple method of bootstrapping development. However, in a production environment, you probably want to configure your own `DataSource`.

If your application uses only a single `DataSource` and that serves as both your business schema and the task repository, all you need to do is provide any `DataSource` (the easiest way to do so is through Spring Boot's configuration conventions). This `DataSource` is automatically used by Spring Cloud Task for the repository.

If your application uses more than one `DataSource`, you need to configure the task repository with the appropriate `DataSource`. This customization can be done through an implementation of `TaskConfigurer`.

172.2. Table Prefix

One modifiable property of `TaskRepository` is the table prefix for the task tables. By default, they are all prefaced with `TASK_`. `TASK_EXECUTION` and `TASK_EXECUTION_PARAMS` are two examples. However, there are potential reasons to modify this prefix. If the schema name needs to be prepended to the table names or if more than one set of task tables is needed within the same schema, you must change the table prefix. You can do so by setting the `spring.cloud.task.tablePrefix` to the prefix you need, as follows:

```
spring.cloud.task.tablePrefix=yourPrefix
```

172.3. Enable/Disable table initialization

In cases where you are creating the task tables and do not wish for Spring Cloud Task to create them at task startup, set the `spring.cloud.task.initialize-enabled` property to `false`, as follows:

```
spring.cloud.task.initialize-enabled=false
```

It defaults to `true`.

The property `spring.cloud.task.initialize.enable` has been deprecated.

172.4. Externally Generated Task ID

In some cases, you may want to allow for the time difference between when a task is requested and when the infrastructure actually launches it. Spring Cloud Task lets you create a `TaskExecution`

when the task is requested. Then pass the execution ID of the generated `TaskExecution` to the task so that it can update the `TaskExecution` through the task's lifecycle.

A `TaskExecution` can be created by calling the `createTaskExecution` method on an implementation of the `TaskRepository` that references the datastore that holds the `TaskExecution` objects.

In order to configure your Task to use a generated `TaskExecutionId`, add the following property:

`spring.cloud.task.executionid=yourtaskId`

172.5. External Task Id

Spring Cloud Task lets you store an external task ID for each `TaskExecution`. An example of this would be a task ID provided by Cloud Foundry when a task is launched on the platform. In order to configure your Task to use a generated `TaskExecutionId`, add the following property:

`spring.cloud.task.external-execution-id=<externalTaskId>`

172.6. Parent Task Id

Spring Cloud Task lets you store a parent task ID for each `TaskExecution`. An example of this would be a task that executes another task or tasks and you want to record which task launched each of the child tasks. In order to configure your Task to set a parent `TaskExecutionId` add the following property on the child task:

`spring.cloud.task.parent-execution-id=<parentExecutionTaskId>`

172.7. TaskConfigurer

The `TaskConfigurer` is a strategy interface that lets you customize the way components of Spring Cloud Task are configured. By default, we provide the `DefaultTaskConfigurer` that provides logical defaults: `Map`-based in-memory components (useful for development if no `DataSource` is provided) and JDBC based components (useful if there is a `DataSource` available).

The `TaskConfigurer` lets you configure three main components:

Component	Description	Default (provided by `DefaultTaskConfigurer`)
`TaskRepository`	The implementation of the `TaskRepository` to be used.	`SimpleTaskRepository`
`TaskExplorer`	The implementation of the `TaskExplorer` (a component for read-only access to the task repository) to be used.	`SimpleTaskExplorer`

Component	Description	Default (provided by `DefaultTaskConfigurer`)
`PlatformTransactionManager`	A transaction manager to be used when running updates for tasks.	`DataSourceTransactionManager` if a `DataSource` is used. `ResourcelessTransactionManager` if it is not.

You can customize any of the components described in the preceding table by creating a custom implementation of the `TaskConfigurer` interface. Typically, extending the `DefaultTaskConfigurer` (which is provided if a `TaskConfigurer` is not found) and overriding the required getter is sufficient. However, implementing your own from scratch may be required.

Users should not directly use getter methods from a `TaskConfigurer` directly unless they are using it to supply implementations to be exposed as Spring Beans.

172.8. Task Name

In most cases, the name of the task is the application name as configured in Spring Boot. However, there are some cases where you may want to map the run of a task to a different name. Spring Cloud Data Flow is an example of this (because you probably want the task to be run with the name of the task definition). Because of this, we offer the ability to customize how the task is named, through the `TaskNameResolver` interface.

By default, Spring Cloud Task provides the `SimpleTaskNameResolver`, which uses the following options (in order of precedence):

1. A Spring Boot property (configured in any of the ways Spring Boot allows) called `spring.cloud.task.name`.
2. The application name as resolved using Spring Boot's rules (obtained through `ApplicationContext#getId`).

172.9. Task Execution Listener

`TaskExecutionListener` lets you register listeners for specific events that occur during the task lifecycle. To do so, create a class that implements the `TaskExecutionListener` interface. The class that implements the `TaskExecutionListener` interface is notified of the following events:

- `onTaskStartup`: Prior to storing the `TaskExecution` into the `TaskRepository`.
- `onTaskEnd`: Prior to updating the `TaskExecution` entry in the `TaskRepository` and marking the final state of the task.
- `onTaskFailed`: Prior to the `onTaskEnd` method being invoked when an unhandled exception is thrown by the task.

Spring Cloud Task also lets you add `TaskExecution` Listeners to methods within a bean by using the following method annotations:

- `@BeforeTask`: Prior to the storing the `TaskExecution` into the `TaskRepository`
- `@AfterTask`: Prior to the updating of the `TaskExecution` entry in the `TaskRepository` marking the final state of the task.
- `@FailedTask`: Prior to the `@AfterTask` method being invoked when an unhandled exception is thrown by the task.

The following example shows the three annotations in use:

```
  public class MyBean {

     @BeforeTask
     public void methodA(TaskExecution taskExecution) {
     }

     @AfterTask
     public void methodB(TaskExecution taskExecution) {
     }

     @FailedTask
     public void methodC(TaskExecution taskExecution, Throwable throwable) {
     }
}
```

Inserting an `ApplicationListener` earlier in the chain than `TaskLifecycleListener` exists may cause unexpected effects.

172.9.1. Exceptions Thrown by Task Execution Listener

If an exception is thrown by a `TaskExecutionListener` event handler, all listener processing for that event handler stops. For example, if three `onTaskStartup` listeners have started and the first `onTaskStartup` event handler throws an exception, the other two `onTaskStartup` methods are not called. However, the other event handlers (`onTaskEnd` and `onTaskFailed`) for the `TaskExecutionListeners` are called.

The exit code returned when a exception is thrown by a `TaskExecutionListener` event handler is the exit code that was reported by the ExitCodeEvent. If no `ExitCodeEvent` is emitted, the Exception thrown is evaluated to see if it is of type ExitCodeGenerator. If so, it returns the exit code from the `ExitCodeGenerator`. Otherwise, `1` is returned.

In the case that an exception is thrown in an `onTaskStartup` method, the exit code for the application will be `1`. If an exception is thrown in either a `onTaskEnd` or `onTaskFailed` method, the exit code for the application will be the one established using the rules enumerated above.

In the case of an exception being thrown in a `onTaskStartup`, `onTaskEnd`, or `onTaskFailed` you can not override the exit code for the application using `ExitCodeExceptionMapper`.

172.9.2. Exit Messages

You can set the exit message for a task programmatically by using a `TaskExecutionListener`. This is done by setting the `TaskExecution’s exitMessage`, which then gets passed into the `TaskExecutionListener`. The following example shows a method that is annotated with the `@AfterTask` `ExecutionListener` :

```
@AfterTask
public void afterMe(TaskExecution taskExecution) {
    taskExecution.setExitMessage("AFTER EXIT MESSAGE");
}
```

An `ExitMessage` can be set at any of the listener events (`onTaskStartup`, `onTaskFailed`, and `onTaskEnd`). The order of precedence for the three listeners follows:

1. `onTaskEnd`
2. `onTaskFailed`
3. `onTaskStartup`

For example, if you set an `exitMessage` for the `onTaskStartup` and `onTaskFailed` listeners and the task ends without failing, the `exitMessage` from the `onTaskStartup` is stored in the repository. Otherwise, if a failure occurs, the `exitMessage` from the `onTaskFailed` is stored. Also if you set the `exitMessage` with an `onTaskEnd` listener, the `exitMessage` from the `onTaskEnd` supersedes the exit messages from both the `onTaskStartup` and `onTaskFailed`.

172.10. Restricting Spring Cloud Task Instances

Spring Cloud Task lets you establish that only one task with a given task name can be run at a time. To do so, you need to establish the task name and set `spring.cloud.task.single-instance-enabled=true` for each task execution. While the first task execution is running, any other time you try to run a task with the same task name and`spring.cloud.task.single-instance-enabled=true`, the task fails with the following error message: `Task with name "application" is already running.` The default value for `spring.cloud.task.single-instance-enabled` is `false`. The following example shows how to set `spring.cloud.task.single-instance-enabled` to `true`:

`spring.cloud.task.single-instance-enabled=true or false`

To use this feature, you must add the following Spring Integration dependencies to your application:

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-core</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-jdbc</artifactId>
</dependency>
```

The exit code for the application will be 1 if the task fails because this feature is enabled and another task is running with the same task name.

172.11. Disabling Spring Cloud Task Auto Configuration

In cases where Spring Cloud Task should not be auto configured for an implementation, you can disable Task's auto configuration. This can be done either by adding the following annotation to your Task application:

```
@EnableAutoConfiguration(exclude={SimpleTaskAutoConfiguration.class})
```

You may also disable Task auto configuration by setting the `spring.cloud.task.autoconfiguration.enabled` property to `false`.

172.12. Closing the Context

If the application requires the `ApplicationContext` to be closed at the completion of a task (all `*Runner#run` methods have been called and the task repository has been updated), set the property `spring.cloud.task.closecontextEnabled` to `true`.

Another case to close the context is when the Task Execution completes however the application does not terminate. In these cases the context is held open because a thread has been allocated (for example: if you are using a TaskExecutor). In these cases set the `spring.cloud.task.closecontextEnabled` property to `true` when launching your task. This will close the application's context once the task is complete. Thus allowing the application to terminate.

Batch

This section goes into more detail about Spring Cloud Task's integration with Spring Batch. Tracking the association between a job execution and the task in which it was executed as well as remote partitioning through Spring Cloud Deployer are covered in this section.

Chapter 173. Associating a Job Execution to the Task in which It Was Executed

Spring Boot provides facilities for the execution of batch jobs within an über-jar. Spring Boot's support of this functionality lets a developer execute multiple batch jobs within that execution. Spring Cloud Task provides the ability to associate the execution of a job (a job execution) with a task's execution so that one can be traced back to the other.

Spring Cloud Task achieves this functionality by using the `TaskBatchExecutionListener`. By default, this listener is auto configured in any context that has both a Spring Batch Job configured (by having a bean of type `Job` defined in the context) and the `spring-cloud-task-batch` jar on the classpath. The listener is injected into all jobs that meet those conditions.

173.1. Overriding the TaskBatchExecutionListener

To prevent the listener from being injected into any batch jobs within the current context, you can disable the autoconfiguration by using standard Spring Boot mechanisms.

To only have the listener injected into particular jobs within the context, override the `batchTaskExecutionListenerBeanPostProcessor` and provide a list of job bean IDs, as shown in the following example:

```
public TaskBatchExecutionListenerBeanPostProcessor
batchTaskExecutionListenerBeanPostProcessor() {
    TaskBatchExecutionListenerBeanPostProcessor postProcessor =
        new TaskBatchExecutionListenerBeanPostProcessor();

    postProcessor.setJobNames(Arrays.asList(new String[] {"job1", "job2"}));

    return postProcessor;
}
```

You can find a sample batch application in the samples module of the Spring Cloud Task Project, here.

Chapter 174. Remote Partitioning

Spring Cloud Deployer provides facilities for launching Spring Boot-based applications on most cloud infrastructures. The `DeployerPartitionHandler` and `DeployerStepExecutionHandler` delegate the launching of worker step executions to Spring Cloud Deployer.

To configure the `DeployerStepExecutionHandler`, you must provide a `Resource` representing the Spring Boot über-jar to be executed, a `TaskLauncher`, and a `JobExplorer`. You can configure any environment properties as well as the max number of workers to be executing at once, the interval to poll for the results (defaults to 10 seconds), and a timeout (defaults to -1 or no timeout). The following example shows how configuring this `PartitionHandler` might look:

```
@Bean
public PartitionHandler partitionHandler(TaskLauncher taskLauncher,
        JobExplorer jobExplorer) throws Exception {

    MavenProperties mavenProperties = new MavenProperties();
    mavenProperties.setRemoteRepositories(new
HashMap<>(Collections.singletonMap("springRepo",
        new MavenProperties.RemoteRepository(repository))));

    Resource resource =
        MavenResource.parse(String.format("%s:%s:%s",
                "io.spring.cloud",
                "partitioned-batch-job",
                "1.1.0.RELEASE"), mavenProperties);

    DeployerPartitionHandler partitionHandler =
        new DeployerPartitionHandler(taskLauncher, jobExplorer, resource,
"workerStep");

    List<String> commandLineArgs = new ArrayList<>(3);
    commandLineArgs.add("--spring.profiles.active=worker");
    commandLineArgs.add("--spring.cloud.task.initialize.enable=false");
    commandLineArgs.add("--spring.batch.initializer.enabled=false");

    partitionHandler.setCommandLineArgsProvider(
        new PassThroughCommandLineArgsProvider(commandLineArgs));
    partitionHandler.setEnvironmentVariablesProvider(new
NoOpEnvironmentVariablesProvider());
    partitionHandler.setMaxWorkers(2);
    partitionHandler.setApplicationName("PartitionedBatchJobTask");

    return partitionHandler;
}
```

When passing environment variables to partitions, each partition may be on a different machine with different environment settings. Consequently, you should pass only those environment variables that are required.

Notice in the example above that we have set the maximum number of workers to 2. Setting the maximum of workers establishes the maximum number of partitions that should be running at one time.

The `Resource` to be executed is expected to be a Spring Boot über-jar with a `DeployerStepExecutionHandler` configured as a `CommandLineRunner` in the current context. The repository enumerated in the preceding example should be the remote repository in which the über-jar is located. Both the manager and worker are expected to have visibility into the same data store being used as the job repository and task repository. Once the underlying infrastructure has bootstrapped the Spring Boot jar and Spring Boot has launched the `DeployerStepExecutionHandler`, the step handler executes the requested `Step`. The following example shows how to configure the `DefaultStepExecutionHandler`:

```
@Bean
public DeployerStepExecutionHandler stepExecutionHandler(JobExplorer jobExplorer) {
    DeployerStepExecutionHandler handler =
        new DeployerStepExecutionHandler(this.context, jobExplorer,
this.jobRepository);

    return handler;
}
```

You can find a sample remote partition application in the samples module of the Spring Cloud Task project, here.

174.1. Notes on Developing a Batch-partitioned application for the Kubernetes Platform

- When deploying partitioned apps on the Kubernetes platform, you must use the following dependency for the Spring Cloud Kubernetes Deployer:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-deployer-kubernetes</artifactId>
</dependency>
```

- The application name for the task application and its partitions need to follow the following regex pattern: `[a-z0-9]([-a-z0-9]*[a-z0-9])`. Otherwise, an exception is thrown.

174.2. Notes on Developing a Batch-partitioned Application for the Cloud Foundry Platform

- When deploying partitioned apps on the Cloud Foundry platform, you must use the following dependencies for the Spring Cloud Foundry Deployer:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-deployer-cloudfoundry</artifactId>
</dependency>
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.1.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>io.projectreactor.ipc</groupId>
    <artifactId>reactor-netty</artifactId>
    <version>0.7.5.RELEASE</version>
</dependency>
```

- When configuring the partition handler, Cloud Foundry Deployment environment variables need to be established so that the partition handler can start the partitions. The following list shows the required environment variables:
  - `spring_cloud_deployer_cloudfoundry_url`
  - `spring_cloud_deployer_cloudfoundry_org`
  - `spring_cloud_deployer_cloudfoundry_space`
  - `spring_cloud_deployer_cloudfoundry_domain`
  - `spring_cloud_deployer_cloudfoundry_username`
  - `spring_cloud_deployer_cloudfoundry_password`
  - `spring_cloud_deployer_cloudfoundry_services`
  - `spring_cloud_deployer_cloudfoundry_taskTimeout`

An example set of deployment environment variables for a partitioned task that uses a `mysql` database service might resemble the following:

```
spring_cloud_deployer_cloudfoundry_url=https://api.local.pcfdev.io
spring_cloud_deployer_cloudfoundry_org=pcfdev-org
spring_cloud_deployer_cloudfoundry_space=pcfdev-space
spring_cloud_deployer_cloudfoundry_domain=local.pcfdev.io
spring_cloud_deployer_cloudfoundry_username=admin
spring_cloud_deployer_cloudfoundry_password=admin
spring_cloud_deployer_cloudfoundry_services=mysql
spring_cloud_deployer_cloudfoundry_taskTimeout=300
```

When using PCF-Dev, the following environment variable is also required: `spring_cloud_deployer_cloudfoundry_skipSslValidation=true`

Chapter 175. Batch Informational Messages

Spring Cloud Task provides the ability for batch jobs to emit informational messages. The "stream.pdf" section covers this feature in detail.

Chapter 176. Batch Job Exit Codes

As discussed earlier, Spring Cloud Task applications support the ability to record the exit code of a task execution. However, in cases where you run a Spring Batch Job within a task, regardless of how the Batch Job Execution completes, the result of the task is always zero when using the default Batch/Boot behavior. Keep in mind that a task is a boot application and that the exit code returned from the task is the same as a boot application. To override this behavior and allow the task to return an exit code other than zero when a batch job returns an BatchStatus of `FAILED`, set `spring.cloud.task.batch.fail-on-job-failure` to `true`. Then the exit code can be 1 (the default) or be based on the specified `ExitCodeGenerator`)

This functionality uses a new `CommandLineRunner` that replaces the one provided by Spring Boot. By default, it is configured with the same order. However, if you want to customize the order in which the `CommandLineRunner` is run, you can set its order by setting the `spring.cloud.task.batch.commandLineRunnerOrder` property. To have your task return the exit code based on the result of the batch job execution, you need to write your own `CommandLineRunner`.

Spring Cloud Stream Integration

A task by itself can be useful, but integration of a task into a larger ecosystem lets it be useful for more complex processing and orchestration. This section covers the integration options for Spring Cloud Task with Spring Cloud Stream.

Chapter 177. Launching a Task from a Spring Cloud Stream

You can launch tasks from a stream. To do so, create a sink that listens for a message that contains a `TaskLaunchRequest` as its payload. The `TaskLaunchRequest` contains:

- `uri`: To the task artifact that is to be executed.
- `applicationName`: The name that is associated with the task. If no applicationName is set, the `TaskLaunchRequest` generates a task name comprised of the following: `Task-<UUID>`.
- `commandLineArguments`: A list containing the command line arguments for the task.
- `environmentProperties`: A map containing the environment variables to be used by the task.
- `deploymentProperties`: A map containing the properties that are used by the deployer to deploy the task.

If the payload is of a different type, the sink throws an exception.

For example, a stream can be created that has a processor that takes in data from an HTTP source and creates a `GenericMessage` that contains the `TaskLaunchRequest` and sends the message to its output channel. The task sink would then receive the message from its input channnel and then launch the task.

To create a taskSink, you need only create a Spring Boot application that includes the `EnableTaskLauncher` annotation, as shown in the following example:

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

The samples module of the Spring Cloud Task project contains a sample Sink and Processor. To install these samples into your local maven repository, run a maven build from the `spring-cloud-task-samples` directory with the `skipInstall` property set to `false`, as shown in the following example:

`mvn clean install`

The `maven.remoteRepositories.springRepo.url` property must be set to the location of the remote repository in which the über-jar is located. If not set, there is no remote repository, so it relies upon the local repository only.

177.1. Spring Cloud Data Flow

To create a stream in Spring Cloud Data Flow, you must first register the Task Sink Application we created. In the following example, we are registering the Processor and Sink sample applications by using the Spring Cloud Data Flow shell:

```
app register --name taskSink --type sink --uri
maven://io.spring.cloud:tasksink:<version>
app register --name taskProcessor --type processor --uri
maven:io.spring.cloud:taskprocessor:<version>
```

The following example shows how to create a stream from the Spring Cloud Data Flow shell:

```
stream create foo --definition "http --server.port=9000|taskProcessor|taskSink"
--deploy
```

Chapter 178. Spring Cloud Task Events

Spring Cloud Task provides the ability to emit events through a Spring Cloud Stream channel when the task is run through a Spring Cloud Stream channel. A task listener is used to publish the `TaskExecution` on a message channel named `task-events`. This feature is autowired into any task that has `spring-cloud-stream`, `spring-cloud-stream-<binder>`, and a defined task on its classpath.

To disable the event emitting listener, set the `spring.cloud.task.events.enabled` property to `false`.

With the appropriate classpath defined, the following task emits the `TaskExecution` as an event on the `task-events` channel (at both the start and the end of the task):

```
@SpringBootApplication
public class TaskEventsApplication {

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

    @Configuration
    public static class TaskConfiguration {

        @Bean
        public CommandLineRunner commandLineRunner() {
            return new CommandLineRunner() {
                @Override
                public void run(String... args) throws Exception {
                    System.out.println("The CommandLineRunner was executed");
                }
            };
        }
    }
}
```

A binder implementation is also required to be on the classpath.

A sample task event application can be found in the samples module of the Spring Cloud Task Project, here.

178.1. Disabling Specific Task Events

To disable task events, you can set the `spring.cloud.task.events.enabled` property to `false`.

Chapter 179. Spring Batch Events

When executing a Spring Batch job through a task, Spring Cloud Task can be configured to emit informational messages based on the Spring Batch listeners available in Spring Batch. Specifically, the following Spring Batch listeners are autoconfigured into each batch job and emit messages on the associated Spring Cloud Stream channels when run through Spring Cloud Task:

- `JobExecutionListener` listens for `job-execution-events`
- `StepExecutionListener` listens for `step-execution-events`
- `ChunkListener` listens for `chunk-events`
- `ItemReadListener` listens for `item-read-events`
- `ItemProcessListener` listens for `item-process-events`
- `ItemWriteListener` listens for `item-write-events`
- `SkipListener` listens for `skip-events`

These listeners are autoconfigured into any `AbstractJob` when the appropriate beans (a `Job` and a `TaskLifecycleListener`) exist in the context. Configuration to listen to these events is handled the same way binding to any other Spring Cloud Stream channel is done. Our task (the one running the batch job) serves as a `Source`, with the listening applications serving as either a `Processor` or a `Sink`.

An example could be to have an application listening to the `job-execution-events` channel for the start and stop of a job. To configure the listening application, you would configure the input to be `job-execution-events` as follows:

`spring.cloud.stream.bindings.input.destination=job-execution-events`

A binder implementation is also required to be on the classpath.

A sample batch event application can be found in the samples module of the Spring Cloud Task Project, here.

179.1. Sending Batch Events to Different Channels

One of the options that Spring Cloud Task offers for batch events is the ability to alter the channel to which a specific listener can emit its messages. To do so, use the following configuration: `spring.cloud.stream.bindings.<the channel>.destination=<new destination>`. For example, if `StepExecutionListener` needs to emit its messages to another channel called `my-step-execution-events` instead of the default `step-execution-events`, you can add the following configuration:

`spring.cloud.stream.bindings.step-execution-events.destination=my-step-execution-events`

179.2. Disabling Batch Events

To disable the listener functionality for all batch events, use the following configuration:

`spring.cloud.task.batch.events.enabled=false`

To disable a specific batch event, use the following configuration:

`spring.cloud.task.batch.events.<batch event listener>.enabled=false`:

The following listing shows individual listeners that you can disable:

```
spring.cloud.task.batch.events.job-execution.enabled=false
spring.cloud.task.batch.events.step-execution.enabled=false
spring.cloud.task.batch.events.chunk.enabled=false
spring.cloud.task.batch.events.item-read.enabled=false
spring.cloud.task.batch.events.item-process.enabled=false
spring.cloud.task.batch.events.item-write.enabled=false
spring.cloud.task.batch.events.skip.enabled=false
```

179.3. Emit Order for Batch Events

By default, batch events have `Ordered.LOWEST_PRECEDENCE`. To change this value (for example, to 5), use the following configuration:

```
spring.cloud.task.batch.events.job-execution-order=5
spring.cloud.task.batch.events.step-execution-order=5
spring.cloud.task.batch.events.chunk-order=5
spring.cloud.task.batch.events.item-read-order=5
spring.cloud.task.batch.events.item-process-order=5
spring.cloud.task.batch.events.item-write-order=5
spring.cloud.task.batch.events.skip-order=5
```

Appendices

Chapter 180. Task Repository Schema

This appendix provides an ERD for the database schema used in the task repository.

[task schema] | *task_schema.png*

180.1. Table Information

TASK_EXECUTION

Stores the task execution information.

Column Name	Required	Type	Field Length	Notes
TASK_EXECUTION_ID	TRUE	BIGINT	X	Spring Cloud Task Framework at app startup establishes the next available id as obtained from the `TASK_SEQ`. Or if the record is created outside of task then the value must be populated at record creation time.
START_TIME	FALSE	DATETIME	X	Spring Cloud Task Framework at app startup establishes the value.
END_TIME	FALSE	DATETIME	X	Spring Cloud Task Framework at app exit establishes the value.
TASK_NAME	FALSE	VARCHAR	100	Spring Cloud Task Framework at app startup will set this to "Application" unless user establish the name using the spring.cloud.task.name as discussed here
EXIT_CODE	FALSE	INTEGER	X	Follows Spring Boot defaults unless overridden by the user as discussed here.
EXIT_MESSAGE	FALSE	VARCHAR	2500	User Defined as discussed here.

Column Name	Required	Type	Field Length	Notes
ERROR_MESSAGE	FALSE	VARCHAR	2500	Spring Cloud Task Framework at app exit establishes the value.
LAST_UPDATED	TRUE	DATETIME	X	Spring Cloud Task Framework at app startup establishes the value. Or if the record is created outside of task then the value must be populated at record creation time.
EXTERNAL_EXECUTION_ID	FALSE	VARCHAR	250	If the `spring.cloud.task.external-execution-id` property is set then Spring Cloud Task Framework at app startup will set this to the value specified. More information can be found here
PARENT_TASK_EXECUTION_ID	FALSE	BIGINT	X	If the `spring.cloud.task.parent-execution-id` property is set then Spring Cloud Task Framework at app startup will set this to the value specified. More information can be found here

TASK_EXECUTION_PARAMS

Stores the parameters used for a task execution

Column Name	Required	Type	Field Length
TASK_EXECUTION_ID	TRUE	BIGINT	X
	TASK_PARAM	FALSE	VARCHAR

TASK_TASK_BATCH

Used to link the task execution to the batch execution.

Column Name	Required	Type	Field Length
TASK_EXECUTION_ID	TRUE	BIGINT	X

Column Name	Required	Type	Field Length
	JOB_EXECUTION_ID	TRUE	BIGINT

TASK_LOCK

Used for the `single-instance-enabled` feature discussed here.

Column Name	Required	Type	Field Length	Notes
LOCK_KEY	TRUE	CHAR	36	UUID for the this lock
REGION	TRUE	VARCHAR	100	User can establish a group of locks using this field.
CLIENT_ID	TRUE	CHAR	36	The task execution id that contains the name of the app to lock.
CREATED_DATE	TRUE	DATETIME	X	The date that the entry was created

The DDL for setting up tables for each database type can be found here.

Chapter 181. Building This Documentation

This project uses Maven to generate this documentation. To generate it for yourself, run the following command: `$ ./mvnw clean package -P full`.

Chapter 182. Running a Task App on Cloud Foundry

The simplest way to launch a Spring Cloud Task application as a task on Cloud Foundry is to use Spring Cloud Data Flow. Via Spring Cloud Data Flow you can register your task application, create a definition for it and then launch it. You then can track the task execution(s) via a RESTful API, the Spring Cloud Data Flow Shell, or the UI. To learn out to get started installing Data Flow follow the instructions in the Getting Started section of the reference documentation. For info on how to register and launch tasks, see the Lifecycle of a Task documentation.

Spring Cloud Vault

© 2016-2020 The original authors.

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.

Spring Cloud Vault Config provides client-side support for externalized configuration in a distributed system. With HashiCorp's Vault you have a central place to manage external secret properties for applications across all environments. Vault can manage static and dynamic secrets such as username/password for remote applications/resources and provide credentials for external services such as MySQL, PostgreSQL, Apache Cassandra, MongoDB, Consul, AWS and more.

Chapter 183. Quick Start

Prerequisites

To get started with Vault and this guide you need a *NIX-like operating systems that provides:

- `wget`, `openssl` and `unzip`
- at least Java 8 and a properly configured `JAVA_HOME` environment variable

This guide explains Vault setup from a Spring Cloud Vault perspective for integration testing. You can find a getting started guide directly on the Vault project site: learn.hashicorp.com/vault

Install Vault

```
$ wget
https://releases.hashicorp.com/vault/${vault_version}/vault_${vault_version}_${platfor
m}.zip
$ unzip vault_${vault_version}_${platform}.zip
```

These steps can be achieved by downloading and running `install_vault.sh`.

Create SSL certificates for Vault

Next, you'r required to generate a set of certificates:

- Root CA
- Vault Certificate (decrypted key `work/ca/private/localhost.decrypted.key.pem` and certificate `work/ca/certs/localhost.cert.pem`)

Make sure to import the Root Certificate into a Java-compliant truststore.

The easiest way to achieve this is by using OpenSSL.

`create_certificates.sh` creates certificates in `work/ca` and a JKS truststore `work/keystore.jks`. If you want to run Spring Cloud Vault using this quickstart guide you need to configure the truststore the `spring.cloud.vault.ssl.trust-store` property to `file:work/keystore.jks`.

Start Vault server

Next create a config file along the lines of:

```
backend "inmem" {
}

listener "tcp" {
  address = "0.0.0.0:8200"
  tls_cert_file = "work/ca/certs/localhost.cert.pem"
  tls_key_file = "work/ca/private/localhost.decrypted.key.pem"
}

disable_mlock = true
```

You can find an example config file at `vault.conf`.

```
$ vault server -config=vault.conf
```

Vault is started listening on `0.0.0.0:8200` using the `inmem` storage and `https`. Vault is sealed and not initialized when starting up.

If you want to run tests, leave Vault uninitialized. The tests will initialize Vault and create a root token `00000000-0000-0000-0000-000000000000`.

If you want to use Vault for your application or give it a try then you need to initialize it first.

```
$ export VAULT_ADDR="https://localhost:8200"
$ export VAULT_SKIP_VERIFY=true # Don't do this for production
$ vault init
```

You should see something like:

```
Key 1: 7149c6a2e16b8833f6eb1e76df03e47f6113a3288b3093faf5033d44f0e70fe701
Key 2: 901c534c7988c18c20435a85213c683bdcf0efcd82e38e2893779f152978c18c02
Key 3: 03ff3948575b1165a20c20ee7c3e6edf04f4cdbe0e82dbff5be49c63f98bc03a03
Key 4: 216ae5cc3ddaf93ceb8e1d15bb9fc3176653f5b738f5f3d1ee00cd7dccbe926e04
Key 5: b2898fc8130929d569c1677ee69dc5f3be57d7c4b494a6062693ce0b1c4d93d805
Initial Root Token: 19aefa97-cccc-bbbb-aaaa-225940e63d76

Vault initialized with 5 keys and a key threshold of 3. Please
securely distribute the above keys. When the Vault is re-sealed,
restarted, or stopped, you must provide at least 3 of these keys
to unseal it again.

Vault does not store the master key. Without at least 3 keys,
your Vault will remain permanently sealed.
```

Vault will initialize and return a set of unsealing keys and the root token. Pick 3 keys and unseal

Vault. Store the Vault token in the `VAULT_TOKEN` environment variable.

```
$ vault unseal (Key 1)
$ vault unseal (Key 2)
$ vault unseal (Key 3)
$ export VAULT_TOKEN=(Root token)
# Required to run Spring Cloud Vault tests after manual initialization
$ vault token-create -id="00000000-0000-0000-0000-000000000000" -policy="root"
```

Spring Cloud Vault accesses different resources. By default, the secret backend is enabled which accesses secret config settings via JSON endpoints.

The HTTP service has resources in the form:

```
/secret/{application}/{profile}
/secret/{application}
/secret/{defaultContext}/{profile}
/secret/{defaultContext}
```

where the "application" is injected as the `spring.application.name` in the `SpringApplication` (i.e. what is normally "application" in a regular Spring Boot app), "profile" is an active profile (or comma-separated list of properties). Properties retrieved from Vault will be used "as-is" without further prefixing of the property names.

Chapter 184. Client Side Usage

To use these features in an application, just build it as a Spring Boot application that depends on `spring-cloud-vault-config` (e.g. see the test cases). Example Maven configuration:

Example 81. pom.xml

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-vault-config</artifactId>
        <version>2.2.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

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

<!-- repositories also needed for snapshots and milestones -->
```

Then you can create a standard Spring Boot application, like this simple HTTP server:

```
@SpringBootApplication
@RestController
public class Application {

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

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

When it runs it will pick up the external configuration from the default local Vault server on port `8200` if it is running. To modify the startup behavior you can change the location of the Vault server using `bootstrap.properties` (like `application.properties` but for the bootstrap phase of an application context), e.g.

Example 82. bootstrap.yml

```
spring.cloud.vault:
    host: localhost
    port: 8200
    scheme: https
    uri: https://localhost:8200
    connection-timeout: 5000
    read-timeout: 15000
    config:
        order: -10
```

- `host` sets the hostname of the Vault host. The host name will be used for SSL certificate validation
- `port` sets the Vault port
- `scheme` setting the scheme to `http` will use plain HTTP. Supported schemes are `http` and `https`.
- `uri` configure the Vault endpoint with an URI. Takes precedence over host/port/scheme configuration
- `connection-timeout` sets the connection timeout in milliseconds
- `read-timeout` sets the read timeout in milliseconds
- `config.order` sets the order for the property source

Enabling further integrations requires additional dependencies and configuration. Depending on how you have set up Vault you might need additional configuration like SSL and authentication.

If the application imports the `spring-boot-starter-actuator` project, the status of the vault server will be available via the `/health` endpoint.

The vault health indicator can be enabled or disabled through the property `management.health.vault.enabled` (default to `true`).

184.1. Authentication

Vault requires an authentication mechanism to authorize client requests.

Spring Cloud Vault supports multiple authentication mechanisms to authenticate applications with Vault.

For a quickstart, use the root token printed by the Vault initialization.

Example 83. bootstrap.yml

```
spring.cloud.vault:
    token: 19aefa97-cccc-bbbb-aaaa-225940e63d76
```

Consider carefully your security requirements. Static token authentication is fine if you want quickly get started with Vault, but a static token is not protected any further. Any disclosure to unintended parties allows Vault use with the associated token roles.

Chapter 185. Authentication methods

Different organizations have different requirements for security and authentication. Vault reflects that need by shipping multiple authentication methods. Spring Cloud Vault supports token and AppId authentication.

185.1. Token authentication

Tokens are the core method for authentication within Vault. Token authentication requires a static token to be provided using the Bootstrap Application Context.

Token authentication is the default authentication method. If a token is disclosed an unintended party gains access to Vault and can access secrets for the intended client.

Example 84. bootstrap.yml

```
spring.cloud.vault:
    authentication: TOKEN
    token: 00000000-0000-0000-0000-000000000000
```

- `authentication` setting this value to `TOKEN` selects the Token authentication method
- `token` sets the static token to use

See also: Vault Documentation: Tokens

185.2. Vault Agent authentication

Vault ships a sidecar utility with Vault Agent since version 0.11.0. Vault Agent implements the functionality of Spring Vault's `SessionManager` with its Auto-Auth feature. Applications can reuse cached session credentials by relying on Vault Agent running on `localhost`. Spring Vault can send requests without the `X-Vault-Token` header. Disable Spring Vault's authentication infrastructure to disable client authentication and session management.

Example 85. bootstrap.yml

```
spring.cloud.vault:
    authentication: NONE
```

- `authentication` setting this value to `NONE` disables `ClientAuthentication` and `SessionManager`.

See also: Vault Documentation: Agent

185.3. AppId authentication

Vault supports AppId authentication that consists of two hard to guess tokens. The AppId defaults to `spring.application.name` that is statically configured. The second token is the UserId which is a part determined by the application, usually related to the runtime environment. IP address, Mac address or a Docker container name are good examples. Spring Cloud Vault Config supports IP address, Mac address and static UserId's (e.g. supplied via System properties). The IP and Mac address are represented as Hex-encoded SHA256 hash.

IP address-based UserId's use the local host's IP address.

Example 86. bootstrap.yml using SHA256 IP-Address UserId's

```
spring.cloud.vault:
    authentication: APPID
    app-id:
        user-id: IP_ADDRESS
```

- `authentication` setting this value to `APPID` selects the AppId authentication method
- `app-id-path` sets the path of the AppId mount to use
- `user-id` sets the UserId method. Possible values are `IP_ADDRESS`, `MAC_ADDRESS` or a class name implementing a custom `AppIdUserIdMechanism`

The corresponding command to generate the IP address UserId from a command line is:

```
$ echo -n 192.168.99.1 | sha256sum
```

Including the line break of `echo` leads to a different hash value so make sure to include the `-n` flag.

Mac address-based UserId's obtain their network device from the localhost-bound device. The configuration also allows specifying a `network-interface` hint to pick the right device. The value of `network-interface` is optional and can be either an interface name or interface index (0-based).

Example 87. bootstrap.yml using SHA256 Mac-Address UserId's

```
spring.cloud.vault:
    authentication: APPID
    app-id:
        user-id: MAC_ADDRESS
        network-interface: eth0
```

- `network-interface` sets network interface to obtain the physical address

The corresponding command to generate the IP address UserId from a command line is:

```
$ echo -n 0AFEDE1234AC | sha256sum
```

The Mac address is specified uppercase and without colons. Including the line break of `echo` leads to a different hash value so make sure to include the `-n` flag.

185.3.1. Custom UserId

The UserId generation is an open mechanism. You can set `spring.cloud.vault.app-id.user-id` to any string and the configured value will be used as static UserId.

A more advanced approach lets you set `spring.cloud.vault.app-id.user-id` to a classname. This class must be on your classpath and must implement the `org.springframework.cloud.vault.AppIdUserIdMechanism` interface and the `createUserId` method. Spring Cloud Vault will obtain the UserId by calling `createUserId` each time it authenticates using AppId to obtain a token.

Example 88. bootstrap.yml

```
spring.cloud.vault:
    authentication: APPID
    app-id:
        user-id: com.examlple.MyUserIdMechanism
```

Example 89. MyUserIdMechanism.java

```
public class MyUserIdMechanism implements AppIdUserIdMechanism {

  @Override
  public String createUserId() {
    String userId = ...
    return userId;
  }
}
```

See also: Vault Documentation: Using the App ID auth backend

185.4. AppRole authentication

AppRole is intended for machine authentication, like the deprecated (since Vault 0.6.1) AppId authentication. AppRole authentication consists of two hard to guess (secret) tokens: RoleId and SecretId.

Spring Vault supports various AppRole scenarios (push/pull mode and wrapped).

RoleId and optionally SecretId must be provided by configuration, Spring Vault will not look up these or create a custom SecretId.

Example 90. bootstrap.yml with AppRole authentication properties

```
spring.cloud.vault:
    authentication: APPROLE
    app-role:
        role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52
```

The following scenarios are supported along the required configuration details:

Table 11. Configuration

Method	RoleId	SecretId	RoleName	Token
Provided RoleId/SecretId	Provided	Provided		
Provided RoleId without SecretId	Provided			
Provided RoleId, Pull SecretId	Provided	Provided	Provided	Provided
Pull RoleId, provided SecretId		Provided	Provided	Provided
Full Pull Mode			Provided	Provided
Wrapped				Provided
Wrapped RoleId, provided SecretId	Provided			Provided
Provided RoleId, wrapped SecretId		Provided		Provided

Table 12. Pull/Push/Wrapped Matrix

RoleId	SecretId	Supported
Provided	Provided	
Provided	Pull	
Provided	Wrapped	
Provided	Absent	
Pull	Provided	
Pull	Pull	
Pull	Wrapped	

Pull	Absent	
Wrapped	Provided	
Wrapped	Pull	
Wrapped	Wrapped	
Wrapped	Absent	

You can use still all combinations of push/pull/wrapped modes by providing a configured `AppRoleAuthentication` bean within the bootstrap context. Spring Cloud Vault cannot derive all possible AppRole combinations from the configuration properties.

AppRole authentication is limited to simple pull mode using reactive infrastructure. Full pull mode is not yet supported. Using Spring Cloud Vault with the Spring WebFlux stack enables Vault's reactive auto-configuration which can be disabled by setting `spring.cloud.vault.reactive.enabled=false`.

Example 91. bootstrap.yml with all AppRole authentication properties

```
spring.cloud.vault:
    authentication: APPROLE
    app-role:
        role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52
        secret-id: 1696536f-1976-73b1-b241-0b4213908d39
        role: my-role
        app-role-path: approle
```

- `role-id` sets the RoleId.
- `secret-id` sets the SecretId. SecretId can be omitted if AppRole is configured without requiring SecretId (See `bind_secret_id`).
- `role`: sets the AppRole name for pull mode.
- `app-role-path` sets the path of the approle authentication mount to use.

See also: Vault Documentation: Using the AppRole auth backend

185.5. AWS-EC2 authentication

The aws-ec2 auth backend provides a secure introduction mechanism for AWS EC2 instances, allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats AWS as a Trusted Third Party and uses the cryptographically signed dynamic metadata information that uniquely represents each EC2 instance.

Example 92. bootstrap.yml using AWS-EC2 Authentication

```
spring.cloud.vault:
    authentication: AWS_EC2
```

AWS-EC2 authentication enables nonce by default to follow the Trust On First Use (TOFU) principle. Any unintended party that gains access to the PKCS#7 identity metadata can authenticate against Vault.

During the first login, Spring Cloud Vault generates a nonce that is stored in the auth backend aside the instance Id. Re-authentication requires the same nonce to be sent. Any other party does not have the nonce and can raise an alert in Vault for further investigation.

The nonce is kept in memory and is lost during application restart. You can configure a static nonce with `spring.cloud.vault.aws-ec2.nonce`.

AWS-EC2 authentication roles are optional and default to the AMI. You can configure the authentication role by setting the `spring.cloud.vault.aws-ec2.role` property.

Example 93. bootstrap.yml with configured role

```
spring.cloud.vault:
    authentication: AWS_EC2
    aws-ec2:
        role: application-server
```

Example 94. bootstrap.yml with all AWS EC2 authentication properties

```
spring.cloud.vault:
    authentication: AWS_EC2
    aws-ec2:
        role: application-server
        aws-ec2-path: aws-ec2
        identity-document: http://...
        nonce: my-static-nonce
```

- `authentication` setting this value to `AWS_EC2` selects the AWS EC2 authentication method
- `role` sets the name of the role against which the login is being attempted.
- `aws-ec2-path` sets the path of the AWS EC2 mount to use
- `identity-document` sets URL of the PKCS#7 AWS EC2 identity document
- `nonce` used for AWS-EC2 authentication. An empty nonce defaults to nonce generation

See also: Vault Documentation: Using the aws auth backend

185.6. AWS-IAM authentication

The aws backend provides a secure authentication mechanism for AWS IAM roles, allowing the automatic authentication with vault based on the current IAM role of the running application. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats AWS as a Trusted Third Party and uses the 4 pieces of information signed by the caller with their IAM credentials to verify that the caller is indeed using that IAM role.

The current IAM role the application is running in is automatically calculated. If you are running your application on AWS ECS then the application will use the IAM role assigned to the ECS task of the running container. If you are running your application naked on top of an EC2 instance then the IAM role used will be the one assigned to the EC2 instance.

When using the AWS-IAM authentication you must create a role in Vault and assign it to your IAM role. An empty `role` defaults to the friendly name the current IAM role.

Example 95. bootstrap.yml with required AWS-IAM Authentication properties

```
spring.cloud.vault:
    authentication: AWS_IAM
```

Example 96. bootstrap.yml with all AWS-IAM Authentication properties

```
spring.cloud.vault:
    authentication: AWS_IAM
    aws-iam:
        role: my-dev-role
        aws-path: aws
        server-name: some.server.name
        endpoint-uri: https://sts.eu-central-1.amazonaws.com
```

- `role` sets the name of the role against which the login is being attempted. This should be bound to your IAM role. If one is not supplied then the friendly name of the current IAM user will be used as the vault role.
- `aws-path` sets the path of the AWS mount to use
- `server-name` sets the value to use for the `X-Vault-AWS-IAM-Server-ID` header preventing certain types of replay attacks.
- `endpoint-uri` sets the value to use for the AWS STS API used for the `iam_request_url` parameter.

AWS-IAM requires the AWS Java SDK dependency (`com.amazonaws:aws-java-sdk-core`) as the authentication implementation uses AWS SDK types for credentials and request signing.

See also: Vault Documentation: Using the aws auth backend

185.7. Azure MSI authentication

The azure auth backend provides a secure introduction mechanism for Azure VM instances, allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats Azure as a Trusted Third Party and uses the managed service identity and instance metadata information that can be bound to a VM instance.

Example 97. bootstrap.yml with required Azure Authentication properties

```
spring.cloud.vault:
    authentication: AZURE_MSI
    azure-msi:
        role: my-dev-role
```

Example 98. bootstrap.yml with all Azure Authentication properties

```
spring.cloud.vault:
    authentication: AZURE_MSI
    azure-msi:
        role: my-dev-role
        azure-path: azure
```

- `role` sets the name of the role against which the login is being attempted.
- `azure-path` sets the path of the Azure mount to use

Azure MSI authentication fetches environmental details about the virtual machine (subscription Id, resource group, VM name) from the instance metadata service.

See also: Vault Documentation: Using the azure auth backend

185.8. TLS certificate authentication

The `cert` auth backend allows authentication using SSL/TLS client certificates that are either signed by a CA or self-signed.

To enable `cert` authentication you need to:

1. Use SSL, see Vault Client SSL configuration
2. Configure a Java `Keystore` that contains the client certificate and the private key
3. Set the `spring.cloud.vault.authentication` to `CERT`

Example 99. bootstrap.yml

```
spring.cloud.vault:
    authentication: CERT
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: changeit
        cert-auth-path: cert
```

See also: Vault Documentation: Using the Cert auth backend

185.9. Cubbyhole authentication

Cubbyhole authentication uses Vault primitives to provide a secured authentication workflow. Cubbyhole authentication uses tokens as primary login method. An ephemeral token is used to obtain a second, login VaultToken from Vault's Cubbyhole secret backend. The login token is usually longer-lived and used to interact with Vault. The login token will be retrieved from a wrapped response stored at `/cubbyhole/response`.

Creating a wrapped token

Response Wrapping for token creation requires Vault 0.6.0 or higher.

Example 100. Creating and storing tokens

```
$ vault token-create -wrap-ttl="10m"
Key                              Value
---                              -----
wrapping_token:                  397ccb93-ff6c-b17b-9389-380b01ca2645
wrapping_token_ttl:              0h10m0s
wrapping_token_creation_time:    2016-09-18 20:29:48.652957077 +0200 CEST
wrapped_accessor:                46b6aebb-187f-932a-26d7-4f3d86a68319
```

Example 101. bootstrap.yml

```
spring.cloud.vault:
    authentication: CUBBYHOLE
    token: 397ccb93-ff6c-b17b-9389-380b01ca2645
```

See also:

- Vault Documentation: Tokens
- Vault Documentation: Cubbyhole Secret Backend

- Vault Documentation: Response Wrapping

185.10. GCP-GCE authentication

The gcp auth backend allows Vault login by using existing GCP (Google Cloud Platform) IAM and GCE credentials.

GCP GCE (Google Compute Engine) authentication creates a signature in the form of a JSON Web Token (JWT) for a service account. A JWT for a Compute Engine instance is obtained from the GCE metadata service using Instance identification. This API creates a JSON Web Token that can be used to confirm the instance identity.

Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats GCP as a Trusted Third Party and uses the cryptographically signed dynamic metadata information that uniquely represents each GCP service account.

Example 102. bootstrap.yml with required GCP-GCE Authentication properties

```
spring.cloud.vault:
    authentication: GCP_GCE
    gcp-gce:
        role: my-dev-role
```

Example 103. bootstrap.yml with all GCP-GCE Authentication properties

```
spring.cloud.vault:
    authentication: GCP_GCE
    gcp-gce:
        gcp-path: gcp
        role: my-dev-role
        service-account: my-service@projectid.iam.gserviceaccount.com
```

- `role` sets the name of the role against which the login is being attempted.
- `gcp-path` sets the path of the GCP mount to use
- `service-account` allows overriding the service account Id to a specific value. Defaults to the `default` service account.

See also:

- Vault Documentation: Using the GCP auth backend
- GCP Documentation: Verifying the Identity of Instances

185.11. GCP-IAM authentication

The gcp auth backend allows Vault login by using existing GCP (Google Cloud Platform) IAM and GCE credentials.

GCP IAM authentication creates a signature in the form of a JSON Web Token (JWT) for a service account. A JWT for a service account is obtained by calling GCP IAM's `projects.serviceAccounts.signJwt` API. The caller authenticates against GCP IAM and proves thereby its identity. This Vault backend treats GCP as a Trusted Third Party.

IAM credentials can be obtained from either the runtime environment , specifically the `GOOGLE_APPLICATION_CREDENTIALS` environment variable, the Google Compute metadata service, or supplied externally as e.g. JSON or base64 encoded. JSON is the preferred form as it carries the project id and service account identifier required for calling `projects.serviceAccounts.signJwt`.

Example 104. bootstrap.yml with required GCP-IAM Authentication properties

```
spring.cloud.vault:
    authentication: GCP_IAM
    gcp-iam:
        role: my-dev-role
```

Example 105. bootstrap.yml with all GCP-IAM Authentication properties

```
spring.cloud.vault:
    authentication: GCP_IAM
    gcp-iam:
        credentials:
            location: classpath:credentials.json
            encoded-key: e+KApn0=
        gcp-path: gcp
        jwt-validity: 15m
        project-id: my-project-id
        role: my-dev-role
        service-account-id: my-service@projectid.iam.gserviceaccount.com
```

- `role` sets the name of the role against which the login is being attempted.
- `credentials.location` path to the credentials resource that contains Google credentials in JSON format.
- `credentials.encoded-key` the base64 encoded contents of an OAuth2 account private key in the JSON format.
- `gcp-path` sets the path of the GCP mount to use
- `jwt-validity` configures the JWT token validity. Defaults to 15 minutes.

- `project-id` allows overriding the project Id to a specific value. Defaults to the project Id from the obtained credential.
- `service-account` allows overriding the service account Id to a specific value. Defaults to the service account from the obtained credential.

GCP IAM authentication requires the Google Cloud Java SDK dependency (`com.google.apis:google-api-services-iam` and `com.google.auth:google-auth-library-oauth2-http`) as the authentication implementation uses Google APIs for credentials and JWT signing.

Google credentials require an OAuth 2 token maintaining the token lifecycle. All API is synchronous therefore, `GcpIamAuthentication` does not support `AuthenticationSteps` which is required for reactive usage.

See also:

- Vault Documentation: Using the GCP auth backend
- GCP Documentation: projects.serviceAccounts.signJwt

185.12. Kubernetes authentication

Kubernetes authentication mechanism (since Vault 0.8.3) allows to authenticate with Vault using a Kubernetes Service Account Token. The authentication is role based and the role is bound to a service account name and a namespace.

A file containing a JWT token for a pod's service account is automatically mounted at `/var/run/secrets/kubernetes.io/serviceaccount/token`.

Example 106. bootstrap.yml with all Kubernetes authentication properties

```
spring.cloud.vault:
    authentication: KUBERNETES
    kubernetes:
        role: my-dev-role
        kubernetes-path: kubernetes
        service-account-token-file:
/var/run/secrets/kubernetes.io/serviceaccount/token
```

- `role` sets the Role.
- `kubernetes-path` sets the path of the Kubernetes mount to use.
- `service-account-token-file` sets the location of the file containing the Kubernetes Service Account Token. Defaults to `/var/run/secrets/kubernetes.io/serviceaccount/token`.

See also:

- Vault Documentation: Kubernetes

- Kubernetes Documentation: Configure Service Accounts for Pods

185.13. Pivotal CloudFoundry authentication

The pcf auth backend provides a secure introduction mechanism for applications running within Pivotal's CloudFoundry instances allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.) as identity provisioning is handled by PCF itself. Instead, it treats PCF as a Trusted Third Party and uses the managed instance identity.

Example 107. bootstrap.yml with required PCF Authentication properties

```
spring.cloud.vault:
    authentication: PCF
    pcf:
        role: my-dev-role
```

Example 108. bootstrap.yml with all PCF Authentication properties

```
spring.cloud.vault:
    authentication: PCF
    pcf:
        role: my-dev-role
        pcf-path: path
        instance-certificate: /etc/cf-instance-credentials/instance.crt
        instance-key: /etc/cf-instance-credentials/instance.key
```

- `role` sets the name of the role against which the login is being attempted.
- `pcf-path` sets the path of the PCF mount to use.
- `instance-certificate` sets the path to the PCF instance identity certificate. Defaults to `${CF_INSTANCE_CERT}` env variable.
- `instance-key` sets the path to the PCF instance identity key. Defaults to `${CF_INSTANCE_KEY}` env variable.

PCF authentication requires BouncyCastle (bcpkix-jdk15on) to be on the classpath for RSA PSS signing.

See also: Vault Documentation: Using the pcf auth backend

Chapter 186. Secret Backends

186.1. Generic Backend

This backend is deprecated in favor of the Key-Value backend and will be removed with the next major version.

Spring Cloud Vault supports at the basic level the key-value secret backend. The key-value secret backend allows storage of arbitrary values as key-value store. A single context can store one or many key-value tuples. Contexts can be organized hierarchically. Spring Cloud Vault allows using the Application name and a default context name (`application`) in combination with active profiles.

```
/secret/{application}/{profile}
/secret/{application}
/secret/{default-context}/{profile}
/secret/{default-context}
```

The application name is determined by the properties:

- `spring.cloud.vault.generic.application-name`
- `spring.cloud.vault.application-name`
- `spring.application.name`

Secrets can be obtained from other contexts within the key-value backend by adding their paths to the application name, separated by commas. For example, given the application name `usefulapp,mysql1,projectx/aws`, each of these folders will be used:

- `/secret/usefulapp`
- `/secret/mysql1`
- `/secret/projectx/aws`

Spring Cloud Vault adds all active profiles to the list of possible context paths. No active profiles will skip accessing contexts with a profile name.

Properties are exposed like they are stored (i.e. without additional prefixes).

```
spring.cloud.vault:
    generic:
        enabled: true
        backend: secret
        profile-separator: '/'
        default-context: application
        application-name: my-app
```

- `enabled` setting this value to `false` disables the secret backend config usage

- `backend` sets the path of the secret mount to use
- `default-context` sets the context name used by all applications
- `application-name` overrides the application name for use in the key-value backend
- `profile-separator` separates the profile name from the context in property sources with profiles

See also: Vault Documentation: Using the KV Secrets Engine - Version 1 (generic secret backend)

186.2. Key-Value Backend

Spring Cloud Vault supports the Key-Value secret backend. The key-value backend allows storage of arbitrary values as key-value store. A single context can store one or many key-value tuples. Contexts can be organized hierarchically. Spring Cloud Vault determines itself whether a secret is using versioning. Spring Cloud Vault allows using the Application name and a default context name (`application`) in combination with active profiles.

```
/secret/{application}/{profile}
/secret/{application}
/secret/{default-context}/{profile}
/secret/{default-context}
```

The application name is determined by the properties:

- `spring.cloud.vault.kv.application-name`
- `spring.cloud.vault.application-name`
- `spring.application.name`

Secrets can be obtained from other contexts within the key-value backend by adding their paths to the application name, separated by commas. For example, given the application name `usefulapp,mysql1,projectx/aws`, each of these folders will be used:

- `/secret/usefulapp`
- `/secret/mysql1`
- `/secret/projectx/aws`

Spring Cloud Vault adds all active profiles to the list of possible context paths. No active profiles will skip accessing contexts with a profile name.

Properties are exposed like they are stored (i.e. without additional prefixes).

Spring Cloud Vault adds the `data/` context between the mount path and the actual context path.

```
spring.cloud.vault:
    kv:
        enabled: true
        backend: secret
        profile-separator: '/'
        default-context: application
        application-name: my-app
```

- `enabled` setting this value to `false` disables the secret backend config usage
- `backend` sets the path of the secret mount to use
- `default-context` sets the context name used by all applications
- `application-name` overrides the application name for use in the key-value backend
- `profile-separator` separates the profile name from the context in property sources with profiles

The key-value secret backend can be operated in versioned (v2) and non-versioned (v1) modes.

See also:

- Vault Documentation: Using the KV Secrets Engine - Version 1 (generic secret backend)
- Vault Documentation: Using the KV Secrets Engine - Version 2 (versioned key-value backend)

186.3. Consul

Spring Cloud Vault can obtain credentials for HashiCorp Consul. The Consul integration requires the `spring-cloud-vault-config-consul` dependency.

Example 109. pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-consul</artifactId>
        <version>2.2.5.RELEASE</version>
    </dependency>
</dependencies>
```

The integration can be enabled by setting `spring.cloud.vault.consul.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.consul.role=…`.

The obtained token is stored in `spring.cloud.consul.token` so using Spring Cloud Consul can pick up the generated credentials without further configuration. You can configure the property name by

setting `spring.cloud.vault.consul.token-property`.

```
spring.cloud.vault:
    consul:
        enabled: true
        role: readonly
        backend: consul
        token-property: spring.cloud.consul.token
```

- `enabled` setting this value to `true` enables the Consul backend config usage
- `role` sets the role name of the Consul role definition
- `backend` sets the path of the Consul mount to use
- `token-property` sets the property name in which the Consul ACL token is stored

See also: [Vault Documentation: Setting up Consul with Vault]

186.4. RabbitMQ

Spring Cloud Vault can obtain credentials for RabbitMQ.

The RabbitMQ integration requires the `spring-cloud-vault-config-rabbitmq` dependency.

Example 110. pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-rabbitmq</artifactId>
        <version>2.2.5.RELEASE</version>
    </dependency>
</dependencies>
```

The integration can be enabled by setting `spring.cloud.vault.rabbitmq.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.rabbitmq.role=…`.

Username and password are stored in `spring.rabbitmq.username` and `spring.rabbitmq.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.rabbitmq.username-property` and `spring.cloud.vault.rabbitmq.password-property`.

```
spring.cloud.vault:
    rabbitmq:
        enabled: true
        role: readonly
        backend: rabbitmq
        username-property: spring.rabbitmq.username
        password-property: spring.rabbitmq.password
```

- `enabled` setting this value to `true` enables the RabbitMQ backend config usage
- `role` sets the role name of the RabbitMQ role definition
- `backend` sets the path of the RabbitMQ mount to use
- `username-property` sets the property name in which the RabbitMQ username is stored
- `password-property` sets the property name in which the RabbitMQ password is stored

See also: Vault Documentation: Setting up RabbitMQ with Vault

186.5. AWS

Spring Cloud Vault can obtain credentials for AWS.

The AWS integration requires the `spring-cloud-vault-config-aws` dependency.

Example 111. pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-aws</artifactId>
        <version>2.2.5.RELEASE</version>
    </dependency>
</dependencies>
```

The integration can be enabled by setting `spring.cloud.vault.aws=true` (default `false`) and providing the role name with `spring.cloud.vault.aws.role=…`.

The access key and secret key are stored in `cloud.aws.credentials.accessKey` and `cloud.aws.credentials.secretKey` so using Spring Cloud AWS will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.aws.access-key-property` and `spring.cloud.vault.aws.secret-key-property`.

```
spring.cloud.vault:
    aws:
        enabled: true
        role: readonly
        backend: aws
        access-key-property: cloud.aws.credentials.accessKey
        secret-key-property: cloud.aws.credentials.secretKey
```

- `enabled` setting this value to `true` enables the AWS backend config usage
- `role` sets the role name of the AWS role definition
- `backend` sets the path of the AWS mount to use
- `access-key-property` sets the property name in which the AWS access key is stored
- `secret-key-property` sets the property name in which the AWS secret key is stored

See also: Vault Documentation: Setting up AWS with Vault

Chapter 187. Database backends

Vault supports several database secret backends to generate database credentials dynamically based on configured roles. This means services that need to access a database no longer need to configure credentials: they can request them from Vault, and use Vault's leasing mechanism to more easily roll keys.

Spring Cloud Vault integrates with these backends:

- Database
- Apache Cassandra
- MongoDB
- MySQL
- PostgreSQL

Using a database secret backend requires to enable the backend in the configuration and the `spring-cloud-vault-config-databases` dependency.

Vault ships since 0.7.1 with a dedicated `database` secret backend that allows database integration via plugins. You can use that specific backend by using the generic database backend. Make sure to specify the appropriate backend path, e.g. `spring.cloud.vault.mysql.role.backend=database`.

Example 112. pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-databases</artifactId>
        <version>2.2.5.RELEASE</version>
    </dependency>
</dependencies>
```

Enabling multiple JDBC-compliant databases will generate credentials and store them by default in the same property keys hence property names for JDBC secrets need to be configured separately.

187.1. Database

Spring Cloud Vault can obtain credentials for any database listed at www.vaultproject.io/api/secret/databases/index.html. The integration can be enabled by setting `spring.cloud.vault.database.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.database.role=…`.

While the database backend is a generic one, `spring.cloud.vault.database` specifically targets JDBC databases. Username and password are stored in `spring.datasource.username` and

`spring.datasource.password` so using Spring Boot will pick up the generated credentials for your `DataSource` without further configuration. You can configure the property names by setting `spring.cloud.vault.database.username-property` and `spring.cloud.vault.database.password-property`.

```
spring.cloud.vault:
    database:
        enabled: true
        role: readonly
        backend: database
        username-property: spring.datasource.username
        password-property: spring.datasource.password
```

- `enabled` setting this value to `true` enables the Database backend config usage
- `role` sets the role name of the Database role definition
- `backend` sets the path of the Database mount to use
- `username-property` sets the property name in which the Database username is stored
- `password-property` sets the property name in which the Database password is stored

See also: Vault Documentation: Database Secrets backend

Spring Cloud Vault does not support getting new credentials and configuring your `DataSource` with them when the maximum lease time has been reached. That is, if `max_ttl` of the Database role in Vault is set to `24h` that means that 24 hours after your application has started it can no longer authenticate with the database.

187.2. Apache Cassandra

The `cassandra` backend has been deprecated in Vault 0.7.1 and it is recommended to use the `database` backend and mount it as `cassandra`.

Spring Cloud Vault can obtain credentials for Apache Cassandra. The integration can be enabled by setting `spring.cloud.vault.cassandra.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.cassandra.role=…`.

Username and password are stored in `spring.data.cassandra.username` and `spring.data.cassandra.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.cassandra.username-property` and `spring.cloud.vault.cassandra.password-property`.

```
spring.cloud.vault:
    cassandra:
        enabled: true
        role: readonly
        backend: cassandra
        username-property: spring.data.cassandra.username
        password-property: spring.data.cassandra.password
```

- `enabled` setting this value to `true` enables the Cassandra backend config usage
- `role` sets the role name of the Cassandra role definition
- `backend` sets the path of the Cassandra mount to use
- `username-property` sets the property name in which the Cassandra username is stored
- `password-property` sets the property name in which the Cassandra password is stored

See also: Vault Documentation: Setting up Apache Cassandra with Vault

187.3. MongoDB

The `mongodb` backend has been deprecated in Vault 0.7.1 and it is recommended to use the `database` backend and mount it as `mongodb`.

Spring Cloud Vault can obtain credentials for MongoDB. The integration can be enabled by setting `spring.cloud.vault.mongodb.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.mongodb.role=…`.

Username and password are stored in `spring.data.mongodb.username` and `spring.data.mongodb.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.mongodb.username-property` and `spring.cloud.vault.mongodb.password-property`.

```
spring.cloud.vault:
    mongodb:
        enabled: true
        role: readonly
        backend: mongodb
        username-property: spring.data.mongodb.username
        password-property: spring.data.mongodb.password
```

- `enabled` setting this value to `true` enables the MongodB backend config usage
- `role` sets the role name of the MongoDB role definition
- `backend` sets the path of the MongoDB mount to use

- `username-property` sets the property name in which the MongoDB username is stored
- `password-property` sets the property name in which the MongoDB password is stored

See also: Vault Documentation: Setting up MongoDB with Vault

187.4. MySQL

The `mysql` backend has been deprecated in Vault 0.7.1 and it is recommended to use the `database` backend and mount it as `mysql`. Configuration for `spring.cloud.vault.mysql` will be removed in a future version.

Spring Cloud Vault can obtain credentials for MySQL. The integration can be enabled by setting `spring.cloud.vault.mysql.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.mysql.role=…`.

Username and password are stored in `spring.datasource.username` and `spring.datasource.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.mysql.username-property` and `spring.cloud.vault.mysql.password-property`.

```
spring.cloud.vault:
    mysql:
        enabled: true
        role: readonly
        backend: mysql
        username-property: spring.datasource.username
        password-property: spring.datasource.password
```

- `enabled` setting this value to `true` enables the MySQL backend config usage
- `role` sets the role name of the MySQL role definition
- `backend` sets the path of the MySQL mount to use
- `username-property` sets the property name in which the MySQL username is stored
- `password-property` sets the property name in which the MySQL password is stored

See also: Vault Documentation: Setting up MySQL with Vault

187.5. PostgreSQL

The `postgresql` backend has been deprecated in Vault 0.7.1 and it is recommended to use the `database` backend and mount it as `postgresql`. Configuration for `spring.cloud.vault.postgresql` will be removed in a future version.

Spring Cloud Vault can obtain credentials for PostgreSQL. The integration can be enabled by setting

`spring.cloud.vault.postgresql.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.postgresql.role=…`.

Username and password are stored in `spring.datasource.username` and `spring.datasource.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.postgresql.username-property` and `spring.cloud.vault.postgresql.password-property`.

```
spring.cloud.vault:
    postgresql:
        enabled: true
        role: readonly
        backend: postgresql
        username-property: spring.datasource.username
        password-property: spring.datasource.password
```

- `enabled` setting this value to `true` enables the PostgreSQL backend config usage
- `role` sets the role name of the PostgreSQL role definition
- `backend` sets the path of the PostgreSQL mount to use
- `username-property` sets the property name in which the PostgreSQL username is stored
- `password-property` sets the property name in which the PostgreSQL password is stored

See also: Vault Documentation: Setting up PostgreSQL with Vault

Chapter 188. Configure `PropertySourceLocator` behavior

Spring Cloud Vault uses property-based configuration to create `PropertySource`s for key-value and discovered secret backends.

Discovered backends provide `VaultSecretBackendDescriptor` beans to describe the configuration state to use secret backend as `PropertySource`. A `SecretBackendMetadataFactory` is required to create a `SecretBackendMetadata` object which contains path, name and property transformation configuration.

`SecretBackendMetadata` is used to back a particular `PropertySource`.

You can register an arbitrary number of beans implementing `VaultConfigurer` for customization. Default key-value and discovered backend registration is disabled if Spring Cloud Vault discovers at least one `VaultConfigurer` bean. You can however enable default registration with `SecretBackendConfigurer.registerDefaultKeyValueSecretBackends()` and `SecretBackendConfigurer.registerDefaultDiscoveredSecretBackends()`.

```
public class CustomizationBean implements VaultConfigurer {

    @Override
    public void addSecretBackends(SecretBackendConfigurer configurer) {

        configurer.add("secret/my-application");

        configurer.registerDefaultKeyValueSecretBackends(false);
        configurer.registerDefaultDiscoveredSecretBackends(true);
    }
}
```

All customization is required to happen in the bootstrap context. Add your configuration classes to `META-INF/spring.factories` at `org.springframework.cloud.bootstrap.BootstrapConfiguration` in your application.

Chapter 189. Service Registry Configuration

You can use a `DiscoveryClient` (such as from Spring Cloud Consul) to locate a Vault server by setting spring.cloud.vault.discovery.enabled=true (default `false`). The net result of that is that your apps need a bootstrap.yml (or an environment variable) with the appropriate discovery configuration. The benefit is that the Vault can change its co-ordinates, as long as the discovery service is a fixed point. The default service id is `vault` but you can change that on the client with `spring.cloud.vault.discovery.serviceId`.

The discovery client implementations all support some kind of metadata map (e.g. for Eureka we have eureka.instance.metadataMap). Some additional properties of the service may need to be configured in its service registration metadata so that clients can connect correctly. Service registries that do not provide details about transport layer security need to provide a `scheme` metadata entry to be set either to `https` or `http`. If no scheme is configured and the service is not exposed as secure service, then configuration defaults to `spring.cloud.vault.scheme` which is `https` when it's not set.

```
spring.cloud.vault.discovery:
    enabled: true
    service-id: my-vault-service
```

Chapter 190. Vault Client Fail Fast

In some cases, it may be desirable to fail startup of a service if it cannot connect to the Vault Server. If this is the desired behavior, set the bootstrap configuration property `spring.cloud.vault.fail-fast=true` and the client will halt with an Exception.

```
spring.cloud.vault:
    fail-fast: true
```

Chapter 191. Vault Enterprise Namespace Support

Vault Enterprise allows using namespaces to isolate multiple Vaults on a single Vault server. Configuring a namespace by setting `spring.cloud.vault.namespace=…` enables the namespace header `X-Vault-Namespace` on every outgoing HTTP request when using the Vault `RestTemplate` or `WebClient`.

Please note that this feature is not supported by Vault Community edition and has no effect on Vault operations.

```
spring.cloud.vault:
    namespace: my-namespace
```

See also: Vault Enterprise: Namespaces

Chapter 192. Vault Client SSL configuration

SSL can be configured declaratively by setting various properties. You can set either `javax.net.ssl.trustStore` to configure JVM-wide SSL settings or `spring.cloud.vault.ssl.trust-store` to set SSL settings only for Spring Cloud Vault Config.

```
spring.cloud.vault:
    ssl:
        trust-store: classpath:keystore.jks
        trust-store-password: changeit
```

- `trust-store` sets the resource for the trust-store. SSL-secured Vault communication will validate the Vault SSL certificate with the specified trust-store.
- `trust-store-password` sets the trust-store password

Please note that configuring `spring.cloud.vault.ssl.*` can be only applied when either Apache Http Components or the OkHttp client is on your class-path.

Chapter 193. Lease lifecycle management (renewal and revocation)

With every secret, Vault creates a lease: metadata containing information such as a time duration, renewability, and more.

Vault promises that the data will be valid for the given duration, or Time To Live (TTL). Once the lease is expired, Vault can revoke the data, and the consumer of the secret can no longer be certain that it is valid.

Spring Cloud Vault maintains a lease lifecycle beyond the creation of login tokens and secrets. That said, login tokens and secrets associated with a lease are scheduled for renewal just before the lease expires until terminal expiry. Application shutdown revokes obtained login tokens and renewable leases.

Secret service and database backends (such as MongoDB or MySQL) usually generate a renewable lease so generated credentials will be disabled on application shutdown.

Static tokens are not renewed or revoked.

Lease renewal and revocation is enabled by default and can be disabled by setting `spring.cloud.vault.config.lifecycle.enabled` to `false`. This is not recommended as leases can expire and Spring Cloud Vault cannot longer access Vault or services using generated credentials and valid credentials remain active after application shutdown.

```
spring.cloud.vault:
    config.lifecycle:
        enabled: true
        min-renewal: 10s
        expiry-threshold: 1m
        lease-endpoints: Legacy
```

- `enabled` controls whether leases associated with secrets are considered to be renewed and expired secrets are rotated. Enabled by default.
- `min-renewal` sets the duration that is at least required before renewing a lease. This setting prevents renewals from happening too often.
- `expiry-threshold` sets the expiry threshold. A lease is renewed the configured period of time before it expires.
- `lease-endpoints` sets the endpoints for renew and revoke. Legacy for vault versions before 0.8 and SysLeases for later.

See also: Vault Documentation: Lease, Renew, and Revoke

Spring Cloud Zookeeper

This project provides Zookeeper integrations for Spring Boot applications through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few annotations, you can quickly enable and configure the common patterns inside your application and build large distributed systems with Zookeeper based components. The provided patterns include Service Discovery and Configuration. Integration with Spring Cloud Netflix provides Intelligent Routing (Zuul), Client Side Load Balancing (Ribbon), and Circuit Breaker (Hystrix).

Chapter 194. Install Zookeeper

See the installation documentation for instructions on how to install Zookeeper.

Spring Cloud Zookeeper uses Apache Curator behind the scenes. While Zookeeper 3.5.x is still considered "beta" by the Zookeeper development team, the reality is that it is used in production by many users. However, Zookeeper 3.4.x is also used in production. Prior to Apache Curator 4.0, both versions of Zookeeper were supported via two versions of Apache Curator. Starting with Curator 4.0 both versions of Zookeeper are supported via the same Curator libraries.

In case you are integrating with version 3.4 you need to change the Zookeeper dependency that comes shipped with `curator`, and thus `spring-cloud-zookeeper`. To do so simply exclude that dependency and add the 3.4.x version like shown below.

maven

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-all</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.12</version>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

gradle

```
compile('org.springframework.cloud:spring-cloud-starter-zookeeper-all') {
  exclude group: 'org.apache.zookeeper', module: 'zookeeper'
}
compile('org.apache.zookeeper:zookeeper:3.4.12') {
  exclude group: 'org.slf4j', module: 'slf4j-log4j12'
}
```

Chapter 195. Service Discovery with Zookeeper

Service Discovery is one of the key tenets of a microservice based architecture. Trying to hand-configure each client or some form of convention can be difficult to do and can be brittle. Curator(A Java library for Zookeeper) provides Service Discovery through a Service Discovery Extension. Spring Cloud Zookeeper uses this extension for service registration and discovery.

195.1. Activating

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` enables autoconfiguration that sets up Spring Cloud Zookeeper Discovery.

For web functionality, you still need to include `org.springframework.boot:spring-boot-starter-web`.

When working with version 3.4 of Zookeeper you need to change the way you include the dependency as described here.

195.2. Registering with Zookeeper

When a client registers with Zookeeper, it provides metadata (such as host and port, ID, and name) about itself.

The following example shows a Zookeeper client:

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

The preceding example is a normal Spring Boot application.

If Zookeeper is located somewhere other than `localhost:2181`, the configuration must provide the location of the server, as shown in the following example:

application.yml

```
spring:
  cloud:
    zookeeper:
      connect-string: localhost:2181
```

If you use Spring Cloud Zookeeper Config, the values shown in the preceding example need to be in `bootstrap.yml` instead of `application.yml`.

The default service name, instance ID, and port (taken from the `Environment`) are `${spring.application.name}`, the Spring Context ID, and `${server.port}`, respectively.

Having `spring-cloud-starter-zookeeper-discovery` on the classpath makes the app into both a Zookeeper "service" (that is, it registers itself) and a "client" (that is, it can query Zookeeper to locate other services).

If you would like to disable the Zookeeper Discovery Client, you can set `spring.cloud.zookeeper.discovery.enabled` to `false`.

195.3. Using the DiscoveryClient

Spring Cloud has support for Feign (a REST client builder), Spring `RestTemplate` and Spring WebFlux, using logical service names instead of physical URLs.

You can also use the `org.springframework.cloud.client.discovery.DiscoveryClient`, which provides a simple API for discovery clients that is not specific to Netflix, as shown in the following example:

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri().toString();
    }
    return null;
}
```

Chapter 196. Using Spring Cloud Zookeeper with Spring Cloud Netflix Components

Spring Cloud Netflix supplies useful tools that work regardless of which `DiscoveryClient` implementation you use. Feign, Turbine, Ribbon, and Zuul all work with Spring Cloud Zookeeper.

196.1. Ribbon with Zookeeper

Spring Cloud Zookeeper provides an implementation of Ribbon's `ServerList`. When you use the `spring-cloud-starter-zookeeper-discovery`, Ribbon is autoconfigured to use the `ZookeeperServerList` by default.

Chapter 197. Spring Cloud Zookeeper and Service Registry

Spring Cloud Zookeeper implements the `ServiceRegistry` interface, letting developers register arbitrary services in a programmatic way.

The `ServiceInstanceRegistration` class offers a `builder()` method to create a `Registration` object that can be used by the `ServiceRegistry`, as shown in the following example:

```
@Autowired
private ZookeeperServiceRegistry serviceRegistry;

public void registerThings() {
    ZookeeperRegistration registration = ServiceInstanceRegistration.builder()
            .defaultUriSpec()
            .address("anyUrl")
            .port(10)
            .name("/a/b/c/d/anotherservice")
            .build();
    this.serviceRegistry.register(registration);
}
```

197.1. Instance Status

Netflix Eureka supports having instances that are `OUT_OF_SERVICE` registered with the server. These instances are not returned as active service instances. This is useful for behaviors such as blue/green deployments. (Note that the Curator Service Discovery recipe does not support this behavior.) Taking advantage of the flexible payload has let Spring Cloud Zookeeper implement `OUT_OF_SERVICE` by updating some specific metadata and then filtering on that metadata in the Ribbon `ZookeeperServerList`. The `ZookeeperServerList` filters out all non-null instance statuses that do not equal `UP`. If the instance status field is empty, it is considered to be `UP` for backwards compatibility. To change the status of an instance, make a `POST` with `OUT_OF_SERVICE` to the `ServiceRegistry` instance status actuator endpoint, as shown in the following example:

```
$ http POST http://localhost:8081/service-registry status=OUT_OF_SERVICE
```

The preceding example uses the `http` command from httpie.org.

Chapter 198. Zookeeper Dependencies

The following topics cover how to work with Spring Cloud Zookeeper dependencies:

- Using the Zookeeper Dependencies
- Activating Zookeeper Dependencies
- Setting up Zookeeper Dependencies
- Configuring Spring Cloud Zookeeper Dependencies

198.1. Using the Zookeeper Dependencies

Spring Cloud Zookeeper gives you a possibility to provide dependencies of your application as properties. As dependencies, you can understand other applications that are registered in Zookeeper and which you would like to call through Feign (a REST client builder), Spring `RestTemplate` and Spring WebFlux.

You can also use the Zookeeper Dependency Watchers functionality to control and monitor the state of your dependencies.

198.2. Activating Zookeeper Dependencies

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` enables autoconfiguration that sets up Spring Cloud Zookeeper Dependencies. Even if you provide the dependencies in your properties, you can turn off the dependencies. To do so, set the `spring.cloud.zookeeper.dependency.enabled` property to false (it defaults to `true`).

198.3. Setting up Zookeeper Dependencies

Consider the following example of dependency representation:

application.yml

```
spring.application.name: yourServiceName
spring.cloud.zookeeper:
  dependencies:
    newsletter:
      path: /path/where/newsletter/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate: application/vnd.newsletter.$version+json
      version: v1
      headers:
        header1:
            - value1
        header2:
            - value2
      required: false
      stubs: org.springframework:foo:stubs
    mailing:
      path: /path/where/mailing/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate: application/vnd.mailing.$version+json
      version: v1
      required: true
```

The next few sections go through each part of the dependency one by one. The root property name is `spring.cloud.zookeeper.dependencies`.

198.3.1. Aliases

Below the root property you have to represent each dependency as an alias. This is due to the constraints of Ribbon, which requires that the application ID be placed in the URL. Consequently, you cannot pass any complex path, suchas `/myApp/myRoute/name`). The alias is the name you use instead of the `serviceId` for `DiscoveryClient`, `Feign`, or `RestTemplate`.

In the previous examples, the aliases are `newsletter` and `mailing`. The following example shows Feign usage with a `newsletter` alias:

```
@FeignClient("newsletter")
public interface NewsletterService {
        @RequestMapping(method = RequestMethod.GET, value = "/newsletter")
        String getNewsletters();
}
```

198.3.2. Path

The path is represented by the `path` YAML property and is the path under which the dependency is registered under Zookeeper. As described in the previous section, Ribbon operates on URLs. As a result, this path is not compliant with its requirement. That is why Spring Cloud Zookeeper maps

the alias to the proper path.

198.3.3. Load Balancer Type

The load balancer type is represented by `loadBalancerType` YAML property.

If you know what kind of load-balancing strategy has to be applied when calling this particular dependency, you can provide it in the YAML file, and it is automatically applied. You can choose one of the following load balancing strategies:

- STICKY: Once chosen, the instance is always called.
- RANDOM: Picks an instance randomly.
- ROUND_ROBIN: Iterates over instances over and over again.

198.3.4. `Content-Type` Template and Version

The `Content-Type` template and version are represented by the `contentTypeTemplate` and `version` YAML properties.

If you version your API in the `Content-Type` header, you do not want to add this header to each of your requests. Also, if you want to call a new version of the API, you do not want to roam around your code to bump up the API version. That is why you can provide a `contentTypeTemplate` with a special `$version` placeholder. That placeholder will be filled by the value of the `version` YAML property. Consider the following example of a `contentTypeTemplate`:

```
application/vnd.newsletter.$version+json
```

Further consider the following `version`:

```
v1
```

The combination of `contentTypeTemplate` and version results in the creation of a `Content-Type` header for each request, as follows:

```
application/vnd.newsletter.v1+json
```

198.3.5. Default Headers

Default headers are represented by the `headers` map in YAML.

Sometimes, each call to a dependency requires setting up of some default headers. To not do that in code, you can set them up in the YAML file, as shown in the following example `headers` section:

```
headers:
    Accept:
        - text/html
        - application/xhtml+xml
    Cache-Control:
        - no-cache
```

That `headers` section results in adding the `Accept` and `Cache-Control` headers with appropriate list of values in your HTTP request.

198.3.6. Required Dependencies

Required dependencies are represented by `required` property in YAML.

If one of your dependencies is required to be up when your application boots, you can set the `required: true` property in the YAML file.

If your application cannot localize the required dependency during boot time, it throws an exception, and the Spring Context fails to set up. In other words, your application cannot start if the required dependency is not registered in Zookeeper.

You can read more about Spring Cloud Zookeeper Presence Checker later in this document.

198.3.7. Stubs

You can provide a colon-separated path to the JAR containing stubs of the dependency, as shown in the following example:

`stubs: org.springframework:myApp:stubs`

where:

- `org.springframework` is the `groupId`.
- `myApp` is the `artifactId`.
- `stubs` is the classifier. (Note that `stubs` is the default value.)

Because `stubs` is the default classifier, the preceding example is equal to the following example:

`stubs: org.springframework:myApp`

198.4. Configuring Spring Cloud Zookeeper Dependencies

You can set the following properties to enable or disable parts of Zookeeper Dependencies functionalities:

- `spring.cloud.zookeeper.dependencies`: If you do not set this property, you cannot use Zookeeper Dependencies.

* `spring.cloud.zookeeper.dependency.ribbon.enabled` (enabled by default): Ribbon requires either explicit global configuration or a particular one for a dependency. By turning on this property, runtime load balancing strategy resolution is possible, and you can use the `loadBalancerType` section of the Zookeeper Dependencies. The configuration that needs this property has an implementation of `LoadBalancerClient` that delegates to the `ILoadBalancer` presented in the next bullet.
* `spring.cloud.zookeeper.dependency.ribbon.loadbalancer` (enabled by default): Thanks to this property, the custom `ILoadBalancer` knows that the part of the URI passed to Ribbon might actually be the alias that has to be resolved to a proper path in Zookeeper. Without this property, you cannot register applications under nested paths.
* `spring.cloud.zookeeper.dependency.headers.enabled` (enabled by default): This property registers a `RibbonClient` that automatically appends appropriate headers and content types with their versions, as presented in the Dependency configuration. Without this setting, those two parameters do not work.
* `spring.cloud.zookeeper.dependency.resttemplate.enabled` (enabled by default): When enabled, this property modifies the request headers of a `@LoadBalanced`-annotated `RestTemplate` such that it passes headers and content type with the version set in dependency configuration. Without this setting, those two parameters do not work.

Chapter 199. Spring Cloud Zookeeper Dependency Watcher

The Dependency Watcher mechanism lets you register listeners to your dependencies. The functionality is, in fact, an implementation of the `Observator` pattern. When a dependency changes, its state (to either UP or DOWN), some custom logic can be applied.

199.1. Activating

Spring Cloud Zookeeper Dependencies functionality needs to be enabled for you to use the Dependency Watcher mechanism.

199.2. Registering a Listener

To register a listener, you must implement an interface called `org.springframework.cloud.zookeeper.discovery.watcher.DependencyWatcherListener` and register it as a bean. The interface gives you one method:

```
void stateChanged(String dependencyName, DependencyState newState);
```

If you want to register a listener for a particular dependency, the `dependencyName` would be the discriminator for your concrete implementation. `newState` provides you with information about whether your dependency has changed to `CONNECTED` or `DISCONNECTED`.

199.3. Using the Presence Checker

Bound with the Dependency Watcher is the functionality called Presence Checker. It lets you provide custom behavior when your application boots, to react according to the state of your dependencies.

The default implementation of the abstract `org.springframework.cloud.zookeeper.discovery.watcher.presence.DependencyPresenceOnStartupVerifier` class is the `org.springframework.cloud.zookeeper.discovery.watcher.presence.DefaultDependencyPresenceOnStartupVerifier`, which works in the following way.

1. If the dependency is marked us `required` and is not in Zookeeper, when your application boots, it throws an exception and shuts down.
2. If the dependency is not `required`, the `org.springframework.cloud.zookeeper.discovery.watcher.presence.LogMissingDependencyChecker` logs that the dependency is missing at the `WARN` level.

Because the `DefaultDependencyPresenceOnStartupVerifier` is registered only when there is no bean of type `DependencyPresenceOnStartupVerifier`, this functionality can be overridden.

Chapter 200. Distributed Configuration with Zookeeper

Zookeeper provides a hierarchical namespace that lets clients store arbitrary data, such as configuration data. Spring Cloud Zookeeper Config is an alternative to the Config Server and Client. Configuration is loaded into the Spring Environment during the special “bootstrap” phase. Configuration is stored in the `/config` namespace by default. Multiple `PropertySource` instances are created, based on the application’s name and the active profiles, to mimic the Spring Cloud Config order of resolving properties. For example, an application with a name of `testApp` and with the `dev` profile has the following property sources created for it:

- `config/testApp,dev`
- `config/testApp`
- `config/application,dev`
- `config/application`

The most specific property source is at the top, with the least specific at the bottom. Properties in the `config/application` namespace apply to all applications that use zookeeper for configuration. Properties in the `config/testApp` namespace are available only to the instances of the service named `testApp`.

Configuration is currently read on startup of the application. Sending a HTTP `POST` request to `/refresh` causes the configuration to be reloaded. Watching the configuration namespace (which Zookeeper supports) is not currently implemented.

200.1. Activating

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-config` enables autoconfiguration that sets up Spring Cloud Zookeeper Config.

When working with version 3.4 of Zookeeper you need to change the way you include the dependency as described here.

200.2. Customizing

Zookeeper Config may be customized by setting the following properties:

bootstrap.yml

```
spring:
  cloud:
    zookeeper:
      config:
        enabled: true
        root: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- `enabled`: Setting this value to `false` disables Zookeeper Config.
- `root`: Sets the base namespace for configuration values.
- `defaultContext`: Sets the name used by all applications.
- `profileSeparator`: Sets the value of the separator used to separate the profile name in property sources with profiles.

200.3. Access Control Lists (ACLs)

You can add authentication information for Zookeeper ACLs by calling the `addAuthInfo` method of a `CuratorFramework` bean. One way to accomplish this is to provide your own `CuratorFramework` bean, as shown in the following example:

```
@BoostrapConfiguration
public class CustomCuratorFrameworkConfig {

  @Bean
  public CuratorFramework curatorFramework() {
    CuratorFramework curator = new CuratorFramework();
    curator.addAuthInfo("digest", "user:password".getBytes());
    return curator;
  }

}
```

Consult the ZookeeperAutoConfiguration class to see how the `CuratorFramework` bean's default configuration.

Alternatively, you can add your credentials from a class that depends on the existing `CuratorFramework` bean, as shown in the following example:

```
@BoostrapConfiguration
public class DefaultCuratorFrameworkConfig {

  public ZookeeperConfig(CuratorFramework curator) {
    curator.addAuthInfo("digest", "user:password".getBytes());
  }

}
```

The creation of this bean must occur during the boostrapping phase. You can register configuration classes to run during this phase by annotating them with `@BootstrapConfiguration` and including them in a comma-separated list that you set as the value of the `org.springframework.cloud.bootstrap.BootstrapConfiguration` property in the `resources/META-INF/spring.factories` file, as shown in the following example:

resources/META-INF/spring.factories

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
my.project.CustomCuratorFrameworkConfig,\
my.project.DefaultCuratorFrameworkConfig
```

Appendix: Compendium of Configuration Properties

Name	Default	Description
aws.paramstore.default-context	application	
aws.paramstore.enabled	true	Is AWS Parameter Store support enabled.
aws.paramstore.fail-fast	true	Throw exceptions during config lookup if true, otherwise, log warnings.
aws.paramstore.name		Alternative to spring.application.name to use in looking up values in AWS Parameter Store.
aws.paramstore.prefix	/config	Prefix indicating first level for every property. Value must start with a forward slash followed by a valid path segment or be empty. Defaults to "/config".
aws.paramstore.profile-separator	_	
aws.paramstore.region		If region value is not null or empty it will be used in creation of AWSSimpleSystemsManagement.
aws.secretsmanager.default-context	application	
aws.secretsmanager.enabled	true	Is AWS Secrets Manager support enabled.
aws.secretsmanager.fail-fast	true	Throw exceptions during config lookup if true, otherwise, log warnings.
aws.secretsmanager.name		Alternative to spring.application.name to use in looking up values in AWS Secrets Manager.

Name	Default	Description
aws.secretsmanager.prefix	/secret	Prefix indicating first level for every property. Value must start with a forward slash followed by a valid path segment or be empty. Defaults to "/config".
aws.secretsmanager.profile-separator	_	
aws.secretsmanager.region		If region value is not null or empty it will be used in creation of AWSSecretsManager.
cloud.aws.credentials.access-key		The access key to be used with a static provider.
cloud.aws.credentials.instance-profile	true	Configures an instance profile credentials provider with no further configuration.
cloud.aws.credentials.profile-name		The AWS profile name.
cloud.aws.credentials.profile-path		The AWS profile path.
cloud.aws.credentials.secret-key		The secret key to be used with a static provider.
cloud.aws.credentials.use-default-aws-credentials-chain	false	Use the DefaultAWSCredentials Chain instead of configuring a custom credentials chain.
cloud.aws.loader.core-pool-size	1	The core pool size of the Task Executor used for parallel S3 interaction. @see org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor#setCorePoolSize(int)
cloud.aws.loader.max-pool-size		The maximum pool size of the Task Executor used for parallel S3 interaction. @see org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor#setMaxPoolSize(int)
cloud.aws.loader.queue-capacity		The maximum queue capacity for backed up S3 requests. @see org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor#setQueueCapacity(int)

Name	Default	Description
cloud.aws.region.auto	true	Enables automatic region detection based on the EC2 meta data service.
cloud.aws.region.static		
cloud.aws.region.use-default-aws-region-chain	false	Whether default AWS SDK region provider chain should be used when auto is set to true.
cloud.aws.stack.auto	true	Enables the automatic stack name detection for the application.
cloud.aws.stack.name		The name of the manually configured stack name that will be used to retrieve the resources.
eureka.client.eureka-connection-idle-timeout-seconds	30	Indicates how much time (in seconds) that the HTTP connections to eureka server can stay idle before it can be closed. In the AWS environment, it is recommended that the values is 30 seconds or less, since the firewall cleans up the connection information after a few mins leaving the connection hanging in limbo.
eureka.client.eureka-server-connect-timeout-seconds	5	Indicates how long to wait (in seconds) before a connection to eureka server needs to timeout. Note that the connections in the client are pooled by org.apache.http.client.HttpClient and this setting affects the actual connection creation and also the wait time to get the connection from the pool.

Name	Default	Description
eureka.client.eureka-server-d-n-s-name		Gets the DNS name to be queried to get the list of eureka servers.This information is not required if the contract returns the service urls by implementing serviceUrls. The DNS mechanism is used when useDnsForFetchingServiceUrls is set to true and the eureka client expects the DNS to configured a certain way so that it can fetch changing eureka servers dynamically. The changes are effective at runtime.
eureka.client.eureka-server-port		Gets the port to be used to construct the service url to contact eureka server when the list of eureka servers come from the DNS.This information is not required if the contract returns the service urls eurekaServerServiceUrls(String). The DNS mechanism is used when useDnsForFetchingServiceUrls is set to true and the eureka client expects the DNS to configured a certain way so that it can fetch changing eureka servers dynamically. The changes are effective at runtime.
eureka.client.eureka-server-read-timeout-seconds	8	Indicates how long to wait (in seconds) before a read from eureka server needs to timeout.
eureka.client.eureka-server-total-connections	200	Gets the total number of connections that is allowed from eureka client to all eureka servers.
eureka.client.eureka-server-total-connections-per-host	50	Gets the total number of connections that is allowed from eureka client to a eureka server host.

Name	Default	Description
eureka.client.eureka-server-u-r-l-context		Gets the URL context to be used to construct the service url to contact eureka server when the list of eureka servers come from the DNS. This information is not required if the contract returns the service urls from eurekaServerServiceUrls. The DNS mechanism is used when useDnsForFetchingServiceUrls is set to true and the eureka client expects the DNS to configured a certain way so that it can fetch changing eureka servers dynamically. The changes are effective at runtime.
eureka.client.eureka-service-url-poll-interval-seconds	0	Indicates how often(in seconds) to poll for changes to eureka server information. Eureka servers could be added or removed and this setting controls how soon the eureka clients should know about it.
eureka.client.prefer-same-zone-eureka	true	Indicates whether or not this instance should try to use the eureka server in the same zone for latency and/or other reason. Ideally eureka clients are configured to talk to servers in the same zone The changes are effective at runtime at the next registry fetch cycle as specified by registryFetchIntervalSeconds
eureka.client.register-with-eureka	true	Indicates whether or not this instance should register its information with eureka server for discovery by others. In some cases, you do not want your instances to be discovered whereas you just want do discover other instances.
eureka.server.peer-eureka-nodes-update-interval-ms	0	

Name	Default	Description
eureka.server.peer-eureka-status-refresh-time-interval-ms	0	
feign.client.config		
feign.client.default-config	default	
feign.client.default-to-properties	true	
feign.compression.request.enabled	false	Enables the request sent by Feign to be compressed.
feign.compression.request.mime-types	[text/xml, application/xml, application/json]	The list of supported mime types.
feign.compression.request.min-request-size	2048	The minimum threshold content size.
feign.compression.response.enabled	false	Enables the response from Feign to be compressed.
feign.compression.response.useGzipDecoder	false	Enables the default gzip decoder to be used.
feign.httpclient.connection-timeout	2000	
feign.httpclient.connection-timer-repeat	3000	
feign.httpclient.disable-ssl-validation	false	
feign.httpclient.enabled	true	Enables the use of the Apache HTTP Client by Feign.
feign.httpclient.follow-redirects	true	
feign.httpclient.max-connections	200	
feign.httpclient.max-connections-per-route	50	
feign.httpclient.time-to-live	900	
feign.httpclient.time-to-live-unit		
feign.hystrix.enabled	false	If true, an OpenFeign client will be wrapped with a Hystrix circuit breaker.
feign.okhttp.enabled	false	Enables the use of the OK HTTP Client by Feign.

Name	Default	Description
management.endpoint.hystrix.config		Hystrix settings. These are traditionally set using servlet parameters. Refer to the documentation of Hystrix for more details.
management.endpoint.hystrix.stream.enabled	true	Whether to enable the hystrix.stream endpoint.
management.metrics.binders.hystrix.enabled	true	Enables creation of OK Http Client factory beans.
ribbon.eureka.enabled	true	Enables the use of Eureka with Ribbon.
spring.cloud.bus.ack.destination-service		Service that wants to listen to acks. By default null (meaning all services).
spring.cloud.bus.ack.enabled	true	Flag to switch off acks (default on).
spring.cloud.bus.destination	springCloudBus	Name of Spring Cloud Stream destination for messages.
spring.cloud.bus.enabled	true	Flag to indicate that the bus is enabled.
spring.cloud.bus.env.enabled	true	Flag to switch off environment change events (default on).
spring.cloud.bus.id	application	The identifier for this application instance.
spring.cloud.bus.refresh.enabled	true	Flag to switch off refresh events (default on).
spring.cloud.bus.trace.enabled	false	Flag to switch on tracing of acks (default off).
spring.cloud.circuitbreaker.hystrix.enabled	true	Enables auto-configuration of the Hystrix Spring Cloud CircuitBreaker API implementation.
spring.cloud.cloudfoundry.discovery.default-server-port	80	Port to use when no port is defined by ribbon.
spring.cloud.cloudfoundry.discovery.enabled	true	Flag to indicate that discovery is enabled.
spring.cloud.cloudfoundry.discovery.heartbeat-frequency	5000	Frequency in milliseconds of poll for heart beat. The client will poll on this frequency and broadcast a list of service ids.

Name	Default	Description
spring.cloud.cloudfoundry.discovery.internal-domain	apps.internal	Default internal domain when configured to use Native DNS service discovery.
spring.cloud.cloudfoundry.discovery.order	0	Order of the discovery client used by `CompositeDiscoveryClient` for sorting available clients.
spring.cloud.cloudfoundry.discovery.use-container-ip	false	Whether to resolve hostname when BOSH DNS is used. In order to use this feature, spring.cloud.cloudfoundry.discovery.use-dns must be true.
spring.cloud.cloudfoundry.discovery.use-dns	false	Whether to use BOSH DNS for the discovery. In order to use this feature, your Cloud Foundry installation must support Service Discovery.
spring.cloud.cloudfoundry.org		Organization name to initially target.
spring.cloud.cloudfoundry.password		Password for user to authenticate and obtain token.
spring.cloud.cloudfoundry.skip-ssl-validation	false	
spring.cloud.cloudfoundry.space		Space name to initially target.
spring.cloud.cloudfoundry.url		URL of Cloud Foundry API (Cloud Controller).
spring.cloud.cloudfoundry.username		Username to authenticate (usually an email address).
spring.cloud.compatibility-verifier.compatible-boot-versions		Default accepted versions for the Spring Boot dependency. You can set {@code x} for the patch version if you don't want to specify a concrete value. Example: {@code 3.4.x}
spring.cloud.compatibility-verifier.enabled	false	Enables creation of Spring Cloud compatibility verification.

Name	Default	Description
spring.cloud.config.allow-override	true	Flag to indicate that {@link #isOverrideSystemProperties() systemPropertiesOverride} can be used. Set to false to prevent users from changing the default accidentally. Default true.
spring.cloud.config.allow-override	true	Flag to indicate that {@link #isOverrideSystemProperties() systemPropertiesOverride} can be used. Set to false to prevent users from changing the default accidentally. Default true.
spring.cloud.config.discovery.enabled	false	Flag to indicate that config server discovery is enabled (config server URL will be looked up via discovery).
spring.cloud.config.discovery.service-id	configserver	Service id to locate config server.
spring.cloud.config.enabled	true	Flag to say that remote configuration is enabled. Default true;
spring.cloud.config.fail-fast	false	Flag to indicate that failure to connect to the server is fatal (default false).
spring.cloud.config.headers		Additional headers used to create the client request.
spring.cloud.config.label		The label name to use to pull remote configuration properties. The default is set on the server (generally "master" for a git based server).
spring.cloud.config.name		Name of application used to fetch remote properties.
spring.cloud.config.override-none	false	Flag to indicate that when {@link #setAllowOverride(boolean) allowOverride} is true, external properties should take lowest priority and should not override any existing property sources (including local config files). Default false.

Name	Default	Description
spring.cloud.config.override-none	false	Flag to indicate that when {@link #setAllowOverride(boolean) allowOverride} is true, external properties should take lowest priority and should not override any existing property sources (including local config files). Default false.
spring.cloud.config.override-system-properties	true	Flag to indicate that the external properties should override system properties. Default true.
spring.cloud.config.override-system-properties	true	Flag to indicate that the external properties should override system properties. Default true.
spring.cloud.config.password		The password to use (HTTP Basic) when contacting the remote server.
spring.cloud.config.profile	default	The default profile to use when fetching remote configuration (comma-separated). Default is "default".
spring.cloud.config.request-connect-timeout	0	timeout on waiting to connect to the Config Server.
spring.cloud.config.request-read-timeout	0	timeout on waiting to read data from the Config Server.
spring.cloud.config.retry.initial-interval	1000	Initial retry interval in milliseconds.
spring.cloud.config.retry.max-attempts	6	Maximum number of attempts.
spring.cloud.config.retry.max-interval	2000	Maximum interval for backoff.
spring.cloud.config.retry.multiplier	1.1	Multiplier for next interval.
spring.cloud.config.send-state	true	Flag to indicate whether to send state. Default true.
spring.cloud.config.tls		TLS properties.

Name	Default	Description
spring.cloud.config.token		Security Token passed thru to underlying environment repository.
spring.cloud.config.uri	[localhost:8888]	The URI of the remote server (default localhost:8888).
spring.cloud.config.username		The username to use (HTTP Basic) when contacting the remote server.
spring.cloud.consul.config.acl-token		
spring.cloud.consul.config.data-key	data	If format is Format.PROPERTIES or Format.YAML then the following field is used as key to look up consul for configuration.
spring.cloud.consul.config.default-context	application	
spring.cloud.consul.config.enabled	true	
spring.cloud.consul.config.fail-fast	true	Throw exceptions during config lookup if true, otherwise, log warnings.
spring.cloud.consul.config.format		
spring.cloud.consul.config.name		Alternative to spring.application.name to use in looking up values in consul KV.
spring.cloud.consul.config.prefix	config	
spring.cloud.consul.config.profile-separator	,	
spring.cloud.consul.config.watch.delay	1000	The value of the fixed delay for the watch in millis. Defaults to 1000.
spring.cloud.consul.config.watch.enabled	true	If the watch is enabled. Defaults to true.

Name	Default	Description
spring.cloud.consul.config.watch.wait-time	55	The number of seconds to wait (or block) for watch query, defaults to 55. Needs to be less than default ConsulClient (defaults to 60). To increase ConsulClient timeout create a ConsulClient bean with a custom ConsulRawClient with a custom HttpClient.
spring.cloud.consul.discovery.acl-token		
spring.cloud.consul.discovery.catalog-services-watch-delay	1000	The delay between calls to watch consul catalog in millis, default is 1000.
spring.cloud.consul.discovery.catalog-services-watch-timeout	2	The number of seconds to block while watching consul catalog, default is 2.
spring.cloud.consul.discovery.consistency-mode		Consistency mode for health service request.
spring.cloud.consul.discovery.datacenters		Map of serviceId's -> datacenter to query for in server list. This allows looking up services in another datacenters.
spring.cloud.consul.discovery.default-query-tag		Tag to query for in service list if one is not listed in serverListQueryTags.
spring.cloud.consul.discovery.default-zone-metadata-name	zone	Service instance zone comes from metadata. This allows changing the metadata tag name.
spring.cloud.consul.discovery.deregister	true	Disable automatic de-registration of service in consul.
spring.cloud.consul.discovery.enable-tag-override		Enable tag override for the registered service.
spring.cloud.consul.discovery.enabled	true	Is service discovery enabled?
spring.cloud.consul.discovery.fail-fast	true	Throw exceptions during service registration if true, otherwise, log warnings (defaults to true).

Name	Default	Description
spring.cloud.consul.discovery.health-check-critical-timeout		Timeout to deregister services critical for longer than timeout (e.g. 30m). Requires consul version 7.x or higher.
spring.cloud.consul.discovery.health-check-headers		Headers to be applied to the Health Check calls.
spring.cloud.consul.discovery.health-check-interval	10s	How often to perform the health check (e.g. 10s), defaults to 10s.
spring.cloud.consul.discovery.health-check-path	/actuator/health	Alternate server path to invoke for health checking.
spring.cloud.consul.discovery.health-check-timeout		Timeout for health check (e.g. 10s).
spring.cloud.consul.discovery.health-check-tls-skip-verify		Skips certificate verification during service checks if true, otherwise runs certificate verification.
spring.cloud.consul.discovery.health-check-url		Custom health check url to override default.
spring.cloud.consul.discovery.heartbeat.enabled	false	
spring.cloud.consul.discovery.heartbeat.interval-ratio		
spring.cloud.consul.discovery.heartbeat.ttl-unit	s	
spring.cloud.consul.discovery.heartbeat.ttl-value	30	
spring.cloud.consul.discovery.hostname		Hostname to use when accessing server.
spring.cloud.consul.discovery.include-hostname-in-instance-id	false	Whether hostname is included into the default instance id when registering service.
spring.cloud.consul.discovery.instance-group		Service instance group.
spring.cloud.consul.discovery.instance-id		Unique service instance id.
spring.cloud.consul.discovery.instance-zone		Service instance zone.

Name	Default	Description
spring.cloud.consul.discovery.ip-address		IP address to use when accessing service (must also set preferIpAddress to use).
spring.cloud.consul.discovery.lifecycle.enabled	true	
spring.cloud.consul.discovery.management-enable-tag-override		Enable tag override for the registered management service.
spring.cloud.consul.discovery.management-metadata		Metadata to use when registering management service.
spring.cloud.consul.discovery.management-port		Port to register the management service under (defaults to management port).
spring.cloud.consul.discovery.management-suffix	management	Suffix to use when registering management service.
spring.cloud.consul.discovery.management-tags		Tags to use when registering management service.
spring.cloud.consul.discovery.metadata		Metadata to use when registering service.
spring.cloud.consul.discovery.order	0	Order of the discovery client used by `CompositeDiscoveryClient` for sorting available clients.
spring.cloud.consul.discovery.port		Port to register the service under (defaults to listening port).
spring.cloud.consul.discovery.prefer-agent-address	false	Source of how we will determine the address to use.
spring.cloud.consul.discovery.prefer-ip-address	false	Use ip address rather than hostname during registration.
spring.cloud.consul.discovery.query-passing	false	Add the 'passing` parameter to /v1/health/service/serviceName. This pushes health check passing to the server.
spring.cloud.consul.discovery.register	true	Register as a service in consul.
spring.cloud.consul.discovery.register-health-check	true	Register health check in consul. Useful during development of a service.

Name	Default	Description
spring.cloud.consul.discovery.scheme	http	Whether to register an http or https service.
spring.cloud.consul.discovery.server-list-query-tags		Map of serviceId's -> tag to query for in server list. This allows filtering services by a single tag.
spring.cloud.consul.discovery.service-name		Service name.
spring.cloud.consul.discovery.tags		Tags to use when registering service.
spring.cloud.consul.discovery.tags-as-metadata	true	Use tags as metadata, defaults to true.
spring.cloud.consul.enabled	true	Is spring cloud consul enabled.
spring.cloud.consul.host	localhost	Consul agent hostname. Defaults to 'localhost'.
spring.cloud.consul.port	8500	Consul agent port. Defaults to '8500'.
spring.cloud.consul.retry.enabled	true	If consul retry is enabled.
spring.cloud.consul.retry.initial-interval	1000	Initial retry interval in milliseconds.
spring.cloud.consul.retry.max-attempts	6	Maximum number of attempts.
spring.cloud.consul.retry.max-interval	2000	Maximum interval for backoff.
spring.cloud.consul.retry.multiplier	1.1	Multiplier for next interval.
spring.cloud.consul.scheme		Consul agent scheme (HTTP/HTTPS). If there is no scheme in address - client will use HTTP.
spring.cloud.consul.service-registry.auto-registration.enabled	true	Enables Consul Service Registry Auto-registration.
spring.cloud.consul.service-registry.enabled	true	Enables Consul Service Registry functionality.
spring.cloud.consul.tls.certificate-password		Password to open the certificate.

Name	Default	Description
spring.cloud.consul.tls.certificate-path		File path to the certificate.
spring.cloud.consul.tls.key-store-instance-type		Type of key framework to use.
spring.cloud.consul.tls.key-store-password		Password to an external keystore.
spring.cloud.consul.tls.key-store-path		Path to an external keystore.
spring.cloud.discovery.client.composite-indicator.enabled	true	Enables discovery client composite health indicator.
spring.cloud.discovery.client.health-indicator.enabled	true	
spring.cloud.discovery.client.health-indicator.include-description	false	
spring.cloud.discovery.client.simple.instances		
spring.cloud.discovery.client.simple.local.instance-id		The unique identifier or name for the service instance.
spring.cloud.discovery.client.simple.local.metadata		Metadata for the service instance. Can be used by discovery clients to modify their behaviour per instance, e.g. when load balancing.
spring.cloud.discovery.client.simple.local.service-id		The identifier or name for the service. Multiple instances might share the same service ID.
spring.cloud.discovery.client.simple.local.uri		The URI of the service instance. Will be parsed to extract the scheme, host, and port.
spring.cloud.discovery.client.simple.order		
spring.cloud.discovery.enabled	true	Enables discovery client health indicators.
spring.cloud.features.enabled	true	Enables the features endpoint.
spring.cloud.gateway.default-filters		List of filter definitions that are applied to every route.

Name	Default	Description
spring.cloud.gateway.discovery.locator.enabled	false	Flag that enables DiscoveryClient gateway integration.
spring.cloud.gateway.discovery.locator.filters		
spring.cloud.gateway.discovery.locator.include-expression	true	SpEL expression that will evaluate whether to include a service in gateway integration or not, defaults to: true.
spring.cloud.gateway.discovery.locator.lower-case-service-id	false	Option to lower case serviceId in predicates and filters, defaults to false. Useful with eureka when it automatically uppercases serviceId. so MYSERIVCE, would match /myservice/**
spring.cloud.gateway.discovery.locator.predicates		
spring.cloud.gateway.discovery.locator.route-id-prefix		The prefix for the routeId, defaults to discoveryClient.getClass().getSimpleName() + "_". Service Id will be appended to create the routeId.
spring.cloud.gateway.discovery.locator.url-expression	'lb://'+serviceId	SpEL expression that create the uri for each route, defaults to: 'lb://'+serviceId.
spring.cloud.gateway.enabled	true	Enables gateway functionality.
spring.cloud.gateway.fail-on-route-definition-error	true	Option to fail on route definition errors, defaults to true. Otherwise, a warning is logged.
spring.cloud.gateway.filter.remove-hop-by-hop.headers		
spring.cloud.gateway.filter.remove-hop-by-hop.order		
spring.cloud.gateway.filter.request-rate-limiter.deny-empty-key	true	Switch to deny requests if the Key Resolver returns an empty key, defaults to true.

Name	Default	Description
spring.cloud.gateway.filter.request-rate-limiter.empty-key-status-code		HttpStatus to return when denyEmptyKey is true, defaults to FORBIDDEN.
spring.cloud.gateway.filter.secure-headers.content-security-policy	default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; object-src 'none'; script-src https:; style-src 'self' https: 'unsafe-inline'	
spring.cloud.gateway.filter.secure-headers.content-type-options	nosniff	
spring.cloud.gateway.filter.secure-headers.disable		
spring.cloud.gateway.filter.secure-headers.download-options	noopen	
spring.cloud.gateway.filter.secure-headers.frame-options	DENY	
spring.cloud.gateway.filter.secure-headers.permitted-cross-domain-policies	none	
spring.cloud.gateway.filter.secure-headers.referrer-policy	no-referrer	
spring.cloud.gateway.filter.secure-headers.strict-transport-security	max-age=631138519	
spring.cloud.gateway.filter.secure-headers.xss-protection-header	1 ; mode=block	
spring.cloud.gateway.forwarded.enabled	true	Enables the ForwardedHeadersFilter.
spring.cloud.gateway.globalcors.add-to-simple-url-handler-mapping	false	If global CORS config should be added to the URL handler.
spring.cloud.gateway.globalcors.cors-configurations		
spring.cloud.gateway.httpclient.connect-timeout		The connect timeout in millis, the default is 45s.
spring.cloud.gateway.httpclient.max-header-size		The max response header size.
spring.cloud.gateway.httpclient.max-initial-line-length		The max initial line length.

Name	Default	Description
spring.cloud.gateway.httpclient.pool.acquire-timeout		Only for type FIXED, the maximum time in millis to wait for aquiring.
spring.cloud.gateway.httpclient.pool.max-connections		Only for type FIXED, the maximum number of connections before starting pending acquisition on existing ones.
spring.cloud.gateway.httpclient.pool.max-idle-time		Time in millis after which the channel will be closed. If NULL, there is no max idle time.
spring.cloud.gateway.httpclient.pool.max-life-time		Duration after which the channel will be closed. If NULL, there is no max life time.
spring.cloud.gateway.httpclient.pool.name	proxy	The channel pool map name, defaults to proxy.
spring.cloud.gateway.httpclient.pool.type		Type of pool for HttpClient to use, defaults to ELASTIC.
spring.cloud.gateway.httpclient.proxy.host		Hostname for proxy configuration of Netty HttpClient.
spring.cloud.gateway.httpclient.proxy.non-proxy-hosts-pattern		Regular expression (Java) for a configured list of hosts. that should be reached directly, bypassing the proxy
spring.cloud.gateway.httpclient.proxy.password		Password for proxy configuration of Netty HttpClient.
spring.cloud.gateway.httpclient.proxy.port		Port for proxy configuration of Netty HttpClient.
spring.cloud.gateway.httpclient.proxy.username		Username for proxy configuration of Netty HttpClient.
spring.cloud.gateway.httpclient.response-timeout		The response timeout.
spring.cloud.gateway.httpclient.ssl.close-notify-flush-timeout	3000ms	SSL close_notify flush timeout. Default to 3000 ms.
spring.cloud.gateway.httpclient.ssl.close-notify-flush-timeout-millis		

Name	Default	Description
spring.cloud.gateway.httpclient.ssl.close-notify-read-timeout		SSL close_notify read timeout. Default to 0 ms.
spring.cloud.gateway.httpclient.ssl.close-notify-read-timeout-millis		
spring.cloud.gateway.httpclient.ssl.default-configuration-type		The default ssl configuration type. Defaults to TCP.
spring.cloud.gateway.httpclient.ssl.handshake-timeout	10000ms	SSL handshake timeout. Default to 10000 ms
spring.cloud.gateway.httpclient.ssl.handshake-timeout-millis		
spring.cloud.gateway.httpclient.ssl.key-password		Key password, default is same as keyStorePassword.
spring.cloud.gateway.httpclient.ssl.key-store		Keystore path for Netty HttpClient.
spring.cloud.gateway.httpclient.ssl.key-store-password		Keystore password.
spring.cloud.gateway.httpclient.ssl.key-store-provider		Keystore provider for Netty HttpClient, optional field.
spring.cloud.gateway.httpclient.ssl.key-store-type	JKS	Keystore type for Netty HttpClient, default is JKS.
spring.cloud.gateway.httpclient.ssl.trusted-x509-certificates		Trusted certificates for verifying the remote endpoint's certificate.
spring.cloud.gateway.httpclient.ssl.use-insecure-trust-manager	false	Installs the netty InsecureTrustManagerFactory. This is insecure and not suitable for production.
spring.cloud.gateway.httpclient.websocket.max-frame-payload-length		Max frame payload length.
spring.cloud.gateway.httpclient.websocket.proxy-ping	true	Proxy ping frames to downstream services, defaults to true.
spring.cloud.gateway.httpclient.wiretap	false	Enables wiretap debugging for Netty HttpClient.
spring.cloud.gateway.httpserver.wiretap	false	Enables wiretap debugging for Netty HttpServer.
spring.cloud.gateway.loadbalancer.use404	false	

Name	Default	Description
spring.cloud.gateway.metrics.enabled	true	Enables the collection of metrics data.
spring.cloud.gateway.metrics.tags		Tags map that added to metrics.
spring.cloud.gateway.redis-rate-limiter.burst-capacity-header	X-RateLimit-Burst-Capacity	The name of the header that returns the burst capacity configuration.
spring.cloud.gateway.redis-rate-limiter.config		
spring.cloud.gateway.redis-rate-limiter.include-headers	true	Whether or not to include headers containing rate limiter information, defaults to true.
spring.cloud.gateway.redis-rate-limiter.remaining-header	X-RateLimit-Remaining	The name of the header that returns number of remaining requests during the current second.
spring.cloud.gateway.redis-rate-limiter.replenish-rate-header	X-RateLimit-Replenish-Rate	The name of the header that returns the replenish rate configuration.
spring.cloud.gateway.redis-rate-limiter.requested-tokens-header	X-RateLimit-Requested-Tokens	The name of the header that returns the requested tokens configuration.
spring.cloud.gateway.routes		List of Routes.
spring.cloud.gateway.set-status.original-status-header-name		The name of the header which contains http code of the proxied request.
spring.cloud.gateway.streaming-media-types		
spring.cloud.gateway.x-forwarded.enabled	true	If the XForwardedHeadersFilter is enabled.
spring.cloud.gateway.x-forwarded.for-append	true	If appending X-Forwarded-For as a list is enabled.
spring.cloud.gateway.x-forwarded.for-enabled	true	If X-Forwarded-For is enabled.
spring.cloud.gateway.x-forwarded.host-append	true	If appending X-Forwarded-Host as a list is enabled.
spring.cloud.gateway.x-forwarded.host-enabled	true	If X-Forwarded-Host is enabled.
spring.cloud.gateway.x-forwarded.order	0	The order of the XForwardedHeadersFilter.

Name	Default	Description
spring.cloud.gateway.x-forwarded.port-append	true	If appending X-Forwarded-Port as a list is enabled.
spring.cloud.gateway.x-forwarded.port-enabled	true	If X-Forwarded-Port is enabled.
spring.cloud.gateway.x-forwarded.prefix-append	true	If appending X-Forwarded-Prefix as a list is enabled.
spring.cloud.gateway.x-forwarded.prefix-enabled	true	If X-Forwarded-Prefix is enabled.
spring.cloud.gateway.x-forwarded.proto-append	true	If appending X-Forwarded-Proto as a list is enabled.
spring.cloud.gateway.x-forwarded.proto-enabled	true	If X-Forwarded-Proto is enabled.
spring.cloud.gcp.bigquery.credentials.encoded-key		
spring.cloud.gcp.bigquery.credentials.location		
spring.cloud.gcp.bigquery.credentials.scopes		
spring.cloud.gcp.bigquery.dataset-name		Name of the BigQuery dataset to use.
spring.cloud.gcp.bigquery.enabled	true	Auto-configure Google Cloud BigQuery components.
spring.cloud.gcp.bigquery.project-id		Overrides the GCP project ID specified in the Core module to use for BigQuery.
spring.cloud.gcp.config.credentials.encoded-key		
spring.cloud.gcp.config.credentials.location		
spring.cloud.gcp.config.credentials.scopes		
spring.cloud.gcp.config.enabled	true	Auto-configure Google Cloud Runtime components.
spring.cloud.gcp.config.name		Name of the application.

Name	Default	Description
spring.cloud.gcp.config.profile		Comma-delimited string of profiles under which the app is running. Gets its default value from the {@code spring.profiles.active} property, falling back on the {@code spring.profiles.default} property.
spring.cloud.gcp.config.project-id		Overrides the GCP project ID specified in the Core module.
spring.cloud.gcp.config.timeout-millis	60000	Timeout for Google Runtime Configuration API calls.
spring.cloud.gcp.core.enabled	true	Auto-configure Google Cloud Core components.
spring.cloud.gcp.credentials.encoded-key		
spring.cloud.gcp.credentials.location		
spring.cloud.gcp.credentials.scopes		
spring.cloud.gcp.datastore.credentials.encoded-key		
spring.cloud.gcp.datastore.credentials.location		
spring.cloud.gcp.datastore.credentials.scopes		
spring.cloud.gcp.datastore.emulator-host		@deprecated use <code>spring.cloud.gcp.datastore.host</code> instead. @see #host
spring.cloud.gcp.datastore.emulator.consistency	0.9	Consistency to use creating the Datastore server instance. Default: {@code 0.9}
spring.cloud.gcp.datastore.emulator.enabled	false	If enabled the Datastore client will connect to an local datastore emulator.
spring.cloud.gcp.datastore.emulator.port	8081	Is the datastore emulator port. Default: {@code 8081}
spring.cloud.gcp.datastore.enabled	true	Auto-configure Google Cloud Datastore components.

Name	Default	Description
spring.cloud.gcp.datastore.host		The host and port of a Datastore emulator as the following example: localhost:8081.
spring.cloud.gcp.datastore.namespace		
spring.cloud.gcp.datastore.project-id		
spring.cloud.gcp.firestore.credentials.encoded-key		
spring.cloud.gcp.firestore.credentials.location		
spring.cloud.gcp.firestore.credentials.scopes		
spring.cloud.gcp.firestore.emulator.enabled	false	Enables autoconfiguration to use the Firestore emulator.
spring.cloud.gcp.firestore.enabled	true	Auto-configure Google Cloud Firestore components.
spring.cloud.gcp.firestore.host-port	firestore.googleapis.com:443	The host and port of the Firestore emulator service; can be overridden to specify an emulator.
spring.cloud.gcp.firestore.project-id		
spring.cloud.gcp.logging.enabled	true	Auto-configure Google Cloud Stackdriver logging for Spring MVC.
spring.cloud.gcp.metrics.credentials.encoded-key		
spring.cloud.gcp.metrics.credentials.location		
spring.cloud.gcp.metrics.credentials.scopes		
spring.cloud.gcp.metrics.enabled	true	Auto-configure Google Cloud Monitoring for Micrometer.
spring.cloud.gcp.metrics.project-id		Overrides the GCP project ID specified in the Core module.
spring.cloud.gcp.project-id		GCP project ID where services are running.

Name	Default	Description
spring.cloud.gcp.pubsub.credentials.encoded-key		
spring.cloud.gcp.pubsub.credentials.location		
spring.cloud.gcp.pubsub.credentials.scopes		
spring.cloud.gcp.pubsub.emulator-host		The host and port of the local running emulator. If provided, this will setup the client to connect against a running pub/sub emulator.
spring.cloud.gcp.pubsub.enabled	true	Auto-configure Google Cloud Pub/Sub components.
spring.cloud.gcp.pubsub.keep-alive-interval-minutes	5	How often to ping the server to keep the channel alive.
spring.cloud.gcp.pubsub.project-id		Overrides the GCP project ID specified in the Core module.
spring.cloud.gcp.pubsub.publisher.batching.delay-threshold-seconds		The delay threshold to use for batching. After this amount of time has elapsed (counting from the first element added), the elements will be wrapped up in a batch and sent.
spring.cloud.gcp.pubsub.publisher.batching.element-count-threshold		The element count threshold to use for batching.
spring.cloud.gcp.pubsub.publisher.batching.enabled		Enables batching if true.
spring.cloud.gcp.pubsub.publisher.batching.flow-control.limit-exceeded-behavior		The behavior when the specified limits are exceeded.
spring.cloud.gcp.pubsub.publisher.batching.flow-control.max-outstanding-element-count		Maximum number of outstanding elements to keep in memory before enforcing flow control.
spring.cloud.gcp.pubsub.publisher.batching.flow-control.max-outstanding-request-bytes		Maximum number of outstanding bytes to keep in memory before enforcing flow control.

Name	Default	Description
spring.cloud.gcp.pubsub.publisher.batching.request-byte-threshold		The request byte threshold to use for batching.
spring.cloud.gcp.pubsub.publisher.executor-threads	4	Number of threads used by every publisher.
spring.cloud.gcp.pubsub.publisher.retry.initial-retry-delay-seconds		InitialRetryDelay controls the delay before the first retry. Subsequent retries will use this value adjusted according to the RetryDelayMultiplier.
spring.cloud.gcp.pubsub.publisher.retry.initial-rpc-timeout-seconds		InitialRpcTimeout controls the timeout for the initial RPC. Subsequent calls will use this value adjusted according to the RpcTimeoutMultiplier.
spring.cloud.gcp.pubsub.publisher.retry.jittered		Jitter determines if the delay time should be randomized.
spring.cloud.gcp.pubsub.publisher.retry.max-attempts		MaxAttempts defines the maximum number of attempts to perform. If this value is greater than 0, and the number of attempts reaches this limit, the logic will give up retrying even if the total retry time is still lower than TotalTimeout.
spring.cloud.gcp.pubsub.publisher.retry.max-retry-delay-seconds		MaxRetryDelay puts a limit on the value of the retry delay, so that the RetryDelayMultiplier can't increase the retry delay higher than this amount.
spring.cloud.gcp.pubsub.publisher.retry.max-rpc-timeout-seconds		MaxRpcTimeout puts a limit on the value of the RPC timeout, so that the RpcTimeoutMultiplier can't increase the RPC timeout higher than this amount.
spring.cloud.gcp.pubsub.publisher.retry.retry-delay-multiplier		RetryDelayMultiplier controls the change in retry delay. The retry delay of the previous call is multiplied by the RetryDelayMultiplier to calculate the retry delay for the next call.

Name	Default	Description
spring.cloud.gcp.pubsub.publisher.retry.rpc-timeout-multiplier		RpcTimeoutMultiplier controls the change in RPC timeout. The timeout of the previous call is multiplied by the RpcTimeoutMultiplier to calculate the timeout for the next call.
spring.cloud.gcp.pubsub.publisher.retry.total-timeout-seconds		TotalTimeout has ultimate control over how long the logic should keep trying the remote call until it gives up completely. The higher the total timeout, the more retries can be attempted.
spring.cloud.gcp.pubsub.reactive.enabled	true	Auto-configure Google Cloud Pub/Sub Reactive components.
spring.cloud.gcp.pubsub.subscriber.executor-threads	4	Number of threads used by every subscriber.
spring.cloud.gcp.pubsub.subscriber.flow-control.limit-exceeded-behavior		The behavior when the specified limits are exceeded.
spring.cloud.gcp.pubsub.subscriber.flow-control.max-outstanding-element-count		Maximum number of outstanding elements to keep in memory before enforcing flow control.
spring.cloud.gcp.pubsub.subscriber.flow-control.max-outstanding-request-bytes		Maximum number of outstanding bytes to keep in memory before enforcing flow control.
spring.cloud.gcp.pubsub.subscriber.max-ack-extension-period	0	The optional max ack extension period in seconds for the subscriber factory.
spring.cloud.gcp.pubsub.subscriber.max-acknowledgement-threads	4	Number of threads used for batch acknowledgement.
spring.cloud.gcp.pubsub.subscriber.parallel-pull-count		The optional parallel pull count setting for the subscriber factory.
spring.cloud.gcp.pubsub.subscriber.pull-endpoint		The optional pull endpoint setting for the subscriber factory.

Name	Default	Description
spring.cloud.gcp.pubsub.subscriber.retry.initial-retry-delay-seconds		InitialRetryDelay controls the delay before the first retry. Subsequent retries will use this value adjusted according to the RetryDelayMultiplier.
spring.cloud.gcp.pubsub.subscriber.retry.initial-rpc-timeout-seconds		InitialRpcTimeout controls the timeout for the initial RPC. Subsequent calls will use this value adjusted according to the RpcTimeoutMultiplier.
spring.cloud.gcp.pubsub.subscriber.retry.jittered		Jitter determines if the delay time should be randomized.
spring.cloud.gcp.pubsub.subscriber.retry.max-attempts		MaxAttempts defines the maximum number of attempts to perform. If this value is greater than 0, and the number of attempts reaches this limit, the logic will give up retrying even if the total retry time is still lower than TotalTimeout.
spring.cloud.gcp.pubsub.subscriber.retry.max-retry-delay-seconds		MaxRetryDelay puts a limit on the value of the retry delay, so that the RetryDelayMultiplier can't increase the retry delay higher than this amount.
spring.cloud.gcp.pubsub.subscriber.retry.max-rpc-timeout-seconds		MaxRpcTimeout puts a limit on the value of the RPC timeout, so that the RpcTimeoutMultiplier can't increase the RPC timeout higher than this amount.
spring.cloud.gcp.pubsub.subscriber.retry.retry-delay-multiplier		RetryDelayMultiplier controls the change in retry delay. The retry delay of the previous call is multiplied by the RetryDelayMultiplier to calculate the retry delay for the next call.

Name	Default	Description
spring.cloud.gcp.pubsub.subscriber.retry.rpc-timeout-multiplier		RpcTimeoutMultiplier controls the change in RPC timeout. The timeout of the previous call is multiplied by the RpcTimeoutMultiplier to calculate the timeout for the next call.
spring.cloud.gcp.pubsub.subscriber.retry.total-timeout-seconds		TotalTimeout has ultimate control over how long the logic should keep trying the remote call until it gives up completely. The higher the total timeout, the more retries can be attempted.
spring.cloud.gcp.secretmanager.credentials.encoded-key		
spring.cloud.gcp.secretmanager.credentials.location		
spring.cloud.gcp.secretmanager.credentials.scopes		
spring.cloud.gcp.secretmanager.enabled	true	Auto-configure GCP Secret Manager support components.
spring.cloud.gcp.secretmanager.project-id		Overrides the GCP Project ID specified in the Core module.
spring.cloud.gcp.security.firebase.project-id		Overrides the GCP project ID specified in the Core module.
spring.cloud.gcp.security.firebase.public-keys-endpoint	www.googleapis.com/robot/v1/metadata/x509/securetoken@system.gserviceaccount.com	Link to Google's public endpoint containing Firebase public keys.
spring.cloud.gcp.security.iap.algorithm	ES256	Encryption algorithm used to sign the JWK token.
spring.cloud.gcp.security.iap.audience		Non-dynamic audience string to validate.
spring.cloud.gcp.security.iap.enabled	true	Auto-configure Google Cloud IAP identity extraction components.
spring.cloud.gcp.security.iap.header	x-goog-iap-jwt-assertion	Header from which to extract the JWK key.
spring.cloud.gcp.security.iap.issuer	cloud.google.com/iap	JWK issuer to verify.

Name	Default	Description
spring.cloud.gcp.security.iap.registry	www.gstatic.com/iap/verify/public_key-jwk	Link to JWK public key registry.
spring.cloud.gcp.spanner.create-interleaved-table-ddl-on-delete-cascade	true	
spring.cloud.gcp.spanner.credentials.encoded-key		
spring.cloud.gcp.spanner.credentials.location		
spring.cloud.gcp.spanner.credentials.scopes		
spring.cloud.gcp.spanner.database		
spring.cloud.gcp.spanner.emulator-host	localhost:9010	
spring.cloud.gcp.spanner.emulator.enabled	false	Enables auto-configuration to use the Spanner emulator.
spring.cloud.gcp.spanner.enabled	true	Auto-configure Google Cloud Spanner components.
spring.cloud.gcp.spanner.fail-if-pool-exhausted	false	
spring.cloud.gcp.spanner.instance-id		
spring.cloud.gcp.spanner.keep-alive-interval-minutes	-1	
spring.cloud.gcp.spanner.max-idle-sessions	-1	
spring.cloud.gcp.spanner.max-sessions	-1	
spring.cloud.gcp.spanner.min-sessions	-1	
spring.cloud.gcp.spanner.num-rpc-channels	-1	
spring.cloud.gcp.spanner.prefetch-chunks	-1	
spring.cloud.gcp.spanner.project-id		
spring.cloud.gcp.spanner.write-sessions-fraction	-1	

Name	Default	Description
spring.cloud.gcp.sql.credentials		Overrides the GCP OAuth2 credentials specified in the Core module.
spring.cloud.gcp.sql.database-name		Name of the database in the Cloud SQL instance.
spring.cloud.gcp.sql.enabled	true	Auto-configure Google Cloud SQL support components.
spring.cloud.gcp.sql.instance-connection-name		Cloud SQL instance connection name. [GCP_PROJECT_ID]:[INSTANCE_REGION]:[INSTANCE_NAME].
spring.cloud.gcp.storage.auto-create-files		
spring.cloud.gcp.storage.credentials.encoded-key		
spring.cloud.gcp.storage.credentials.location		
spring.cloud.gcp.storage.credentials.scopes		
spring.cloud.gcp.storage.enabled	true	Auto-configure Google Cloud Storage components.
spring.cloud.gcp.storage.project-id		
spring.cloud.gcp.trace.authority		HTTP/2 authority the channel claims to be connecting to.
spring.cloud.gcp.trace.compression		Compression to use for the call.
spring.cloud.gcp.trace.credentials.encoded-key		
spring.cloud.gcp.trace.credentials.location		
spring.cloud.gcp.trace.credentials.scopes		
spring.cloud.gcp.trace.deadline-ms		Call deadline.
spring.cloud.gcp.trace.enabled	true	Auto-configure Google Cloud Stackdriver tracing components.

Name	Default	Description
spring.cloud.gcp.trace.max-inbound-size		Maximum size for an inbound message.
spring.cloud.gcp.trace.max-outbound-size		Maximum size for an outbound message.
spring.cloud.gcp.trace.message-timeout	1	Timeout in seconds before pending spans will be sent in batches to GCP Stackdriver Trace.
spring.cloud.gcp.trace.num-executor-threads	4	Number of threads to be used by the Trace executor.
spring.cloud.gcp.trace.project-id		Overrides the GCP project ID specified in the Core module.
spring.cloud.gcp.trace.wait-for-ready		Waits for the channel to be ready in case of a transient failure. Defaults to failing fast in that case.
spring.cloud.gcp.vision.credentials.encoded-key		
spring.cloud.gcp.vision.credentials.location		
spring.cloud.gcp.vision.credentials.scopes		
spring.cloud.gcp.vision.enabled	true	Auto-configure Google Cloud Vision components.
spring.cloud.gcp.vision.executor-threads-count	1	Number of threads used to poll for the completion of Document OCR operations.
spring.cloud.gcp.vision.json-output-batch-size	20	Number of document pages to include in each JSON output file.
spring.cloud.httpclientfactories.apache.enabled	true	Enables creation of Apache Http Client factory beans.
spring.cloud.httpclientfactories.ok.enabled	true	Enables creation of OK Http Client factory beans.
spring.cloud.hypermedia.refresh.fixed-delay	5000	
spring.cloud.hypermedia.refresh.initial-delay	10000	
spring.cloud.inetutils.default-hostname	localhost	The default hostname. Used in case of errors.

Name	Default	Description
spring.cloud.inetutils.default-ip-address	127.0.0.1	The default IP address. Used in case of errors.
spring.cloud.inetutils.ignored-interfaces		List of Java regular expressions for network interfaces that will be ignored.
spring.cloud.inetutils.preferred-networks		List of Java regular expressions for network addresses that will be preferred.
spring.cloud.inetutils.timeout-seconds	1	Timeout, in seconds, for calculating hostname.
spring.cloud.inetutils.use-only-site-local-interfaces	false	Whether to use only interfaces with site local addresses. See {@link InetAddress#isSiteLocalAddress()} for more details.
spring.cloud.kubernetes.client.api-version		
spring.cloud.kubernetes.client.apiVersion	v1	Kubernetes API Version
spring.cloud.kubernetes.client.ca-cert-data		
spring.cloud.kubernetes.client.ca-cert-file		
spring.cloud.kubernetes.client.caCertData		Kubernetes API CACertData
spring.cloud.kubernetes.client.caCertFile		Kubernetes API CACertFile
spring.cloud.kubernetes.client.client-cert-data		
spring.cloud.kubernetes.client.client-cert-file		
spring.cloud.kubernetes.client.client-key-algo		
spring.cloud.kubernetes.client.client-key-data		
spring.cloud.kubernetes.client.client-key-file		
spring.cloud.kubernetes.client.client-key-passphrase		

Name	Default	Description
spring.cloud.kubernetes.client.clientCertData		Kubernetes API ClientCertData
spring.cloud.kubernetes.client.clientCertFile		Kubernetes API ClientCertFile
spring.cloud.kubernetes.client.clientKeyAlgo	RSA	Kubernetes API ClientKeyAlgo
spring.cloud.kubernetes.client.clientKeyData		Kubernetes API ClientKeyData
spring.cloud.kubernetes.client.clientKeyFile		Kubernetes API ClientKeyFile
spring.cloud.kubernetes.client.clientKeyPassphrase	changeit	Kubernetes API ClientKeyPassphrase
spring.cloud.kubernetes.client.connection-timeout		
spring.cloud.kubernetes.client.connectionTimeout	10s	Connection timeout
spring.cloud.kubernetes.client.http-proxy		
spring.cloud.kubernetes.client.https-proxy		
spring.cloud.kubernetes.client.logging-interval		
spring.cloud.kubernetes.client.loggingInterval	20s	Logging interval
spring.cloud.kubernetes.client.master-url		
spring.cloud.kubernetes.client.masterUrl	kubernetes.default.svc	Kubernetes API Master Node URL
spring.cloud.kubernetes.client.namespace	true	Kubernetes Namespace
spring.cloud.kubernetes.client.no-proxy		
spring.cloud.kubernetes.client.password		Kubernetes API Password
spring.cloud.kubernetes.client.proxy-password		
spring.cloud.kubernetes.client.proxy-username		

Name	Default	Description
spring.cloud.kubernetes.client.request-timeout		
spring.cloud.kubernetes.client.requestTimeout	10s	Request timeout
spring.cloud.kubernetes.client.rolling-timeout		
spring.cloud.kubernetes.client.rollingTimeout	900s	Rolling timeout
spring.cloud.kubernetes.client.trust-certs		
spring.cloud.kubernetes.client.trustCerts	false	Kubernetes API Trust Certificates
spring.cloud.kubernetes.client.username		Kubernetes API Username
spring.cloud.kubernetes.client.watch-reconnect-interval		
spring.cloud.kubernetes.client.watch-reconnect-limit		
spring.cloud.kubernetes.client.watchReconnectInterval	1s	Reconnect Interval
spring.cloud.kubernetes.client.watchReconnectLimit	-1	Reconnect Interval limit retries
spring.cloud.kubernetes.config.enable-api	true	
spring.cloud.kubernetes.config.enabled	true	Enable the ConfigMap property source locator.
spring.cloud.kubernetes.config.name		
spring.cloud.kubernetes.config.namespace		
spring.cloud.kubernetes.config.paths		
spring.cloud.kubernetes.config.sources		
spring.cloud.kubernetes.discovery.all-namespaces	false	If discovering all namespaces.
spring.cloud.kubernetes.discovery.enabled	true	If Kubernetes Discovery is enabled.

Name	Default	Description
spring.cloud.kubernetes.discovery.filter		SpEL expression to filter services AFTER they have been retrieved from the Kubernetes API server.
spring.cloud.kubernetes.discovery.known-secure-ports		Set the port numbers that are considered secure and use HTTPS.
spring.cloud.kubernetes.discovery.metadata.add-annotations	true	When set, the Kubernetes annotations of the services will be included as metadata of the returned ServiceInstance.
spring.cloud.kubernetes.discovery.metadata.add-labels	true	When set, the Kubernetes labels of the services will be included as metadata of the returned ServiceInstance.
spring.cloud.kubernetes.discovery.metadata.add-ports	true	When set, any named Kubernetes service ports will be included as metadata of the returned ServiceInstance.
spring.cloud.kubernetes.discovery.metadata.annotations-prefix		When addAnnotations is set, then this will be used as a prefix to the key names in the metadata map.
spring.cloud.kubernetes.discovery.metadata.labels-prefix		When addLabels is set, then this will be used as a prefix to the key names in the metadata map.
spring.cloud.kubernetes.discovery.metadata.ports-prefix	port.	When addPorts is set, then this will be used as a prefix to the key names in the metadata map.
spring.cloud.kubernetes.discovery.order		
spring.cloud.kubernetes.discovery.primary-port-name		If set then the port with a given name is used as primary when multiple ports are defined for a service.
spring.cloud.kubernetes.discovery.service-labels		If set, then only the services matching these labels will be fetched from the Kubernetes API server.

Name	Default	Description
spring.cloud.kubernetes.discovery.service-name	unknown	The service name of the local instance.
spring.cloud.kubernetes.enabled	true	Whether to enable Kubernetes integration.
spring.cloud.kubernetes.loadbalancer.cluster-domain	cluster.local	cluster domain.
spring.cloud.kubernetes.loadbalancer.enabled	true	Load balancer enabled,default true.
spring.cloud.kubernetes.loadbalancer.mode		{@link KubernetesLoadBalancerMode} setting load balancer server list with ip of pod or service name. default value is POD.
spring.cloud.kubernetes.loadbalancer.port-name	http	service port name.
spring.cloud.kubernetes.reload.enabled	false	Enables the Kubernetes configuration reload on change.
spring.cloud.kubernetes.reload.max-wait-for-restart	2s	If Restart or Shutdown strategies are used, Spring Cloud Kubernetes waits a random amount of time before restarting. This is done in order to avoid having all instances of the same application restart at the same time. This property configures the maximum of amount of wait time from the moment the signal is received that a restart is needed until the moment the restart is actually triggered
spring.cloud.kubernetes.reload.mode		Sets the detection mode for Kubernetes configuration reload.
spring.cloud.kubernetes.reload.monitoring-config-maps	true	Enables monitoring on config maps to detect changes.
spring.cloud.kubernetes.reload.monitoring-secrets	false	Enables monitoring on secrets to detect changes.
spring.cloud.kubernetes.reload.period	15000ms	Sets the polling period to use when the detection mode is POLLING.

Name	Default	Description
spring.cloud.kubernetes.reload.strategy		Sets the reload strategy for Kubernetes configuration reload on change.
spring.cloud.kubernetes.ribbon.cluster-domain	cluster.local	cluster domain.
spring.cloud.kubernetes.ribbon.enabled	true	Ribbon enabled,default true.
spring.cloud.kubernetes.ribbon.mode		{@link KubernetesRibbonMode} setting ribbon server list with ip of pod or service name. default value is POD.
spring.cloud.kubernetes.secrets.enable-api	false	
spring.cloud.kubernetes.secrets.enabled	true	Enable the Secrets property source locator.
spring.cloud.kubernetes.secrets.labels		
spring.cloud.kubernetes.secrets.name		
spring.cloud.kubernetes.secrets.namespace		
spring.cloud.kubernetes.secrets.paths		
spring.cloud.kubernetes.secrets.sources		
spring.cloud.loadbalancer.cache.caffeine.spec		The spec to use to create caches. See CaffeineSpec for more details on the spec format.
spring.cloud.loadbalancer.cache.capacity	256	Initial cache capacity expressed as int.
spring.cloud.loadbalancer.cache.enabled	true	Enables Spring Cloud LoadBalancer caching mechanism.

Name	Default	Description
spring.cloud.loadbalancer.cache.ttl	35s	Time To Live - time counted from writing of the record, after which cache entries are expired, expressed as a {@link Duration}. The property {@link String} has to be in keeping with the appropriate syntax as specified in Spring Boot <code>StringToDurationConverter</code>. @see github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/convert/StringToDurationConverter.java">StringToDurationConverter.java;
spring.cloud.loadbalancer.eureka.approximate-zone-from-hostname	false	Used to determine whether we should try to get the `zone` value from host name.
spring.cloud.loadbalancer.health-check.initial-delay	0	Initial delay value for the HealthCheck scheduler.
spring.cloud.loadbalancer.health-check.interval	25s	Interval for rerunning the HealthCheck scheduler.
spring.cloud.loadbalancer.health-check.path		
spring.cloud.loadbalancer.retry.enabled	true	
spring.cloud.loadbalancer.ribbon.enabled	true	Causes `RibbonLoadBalancerClient` to be used by default.

Name	Default	Description
spring.cloud.loadbalancer.ribbon.enabled	true	Causes `RibbonLoadBalancerClient` to be used by default.
spring.cloud.loadbalancer.service-discovery.timeout		String representation of Duration of the timeout for calls to service discovery.
spring.cloud.loadbalancer.zone		Spring Cloud LoadBalancer zone.
spring.cloud.refresh.enabled	true	Enables autoconfiguration for the refresh scope and associated features.
spring.cloud.refresh.extra-refreshable	true	Additional class names for beans to post process into refresh scope.
spring.cloud.refresh.never-refreshable	true	Comma separated list of class names for beans to never be refreshed or rebound.
spring.cloud.service-registry.auto-registration.enabled	true	Whether service auto-registration is enabled. Defaults to true.
spring.cloud.service-registry.auto-registration.fail-fast	false	Whether startup fails if there is no AutoServiceRegistration. Defaults to false.
spring.cloud.service-registry.auto-registration.register-management	true	Whether to register the management as a service. Defaults to true.
spring.cloud.util.enabled	true	Enables creation of Spring Cloud utility beans.
spring.cloud.vault.app-id.app-id-path	app-id	Mount path of the AppId authentication backend.
spring.cloud.vault.app-id.network-interface		Network interface hint for the "MAC_ADDRESS" UserId mechanism.
spring.cloud.vault.app-id.user-id	MAC_ADDRESS	UserId mechanism. Can be either "MAC_ADDRESS", "IP_ADDRESS", a string or a class name.
spring.cloud.vault.app-role.app-role-path	approle	Mount path of the AppRole authentication backend.

Name	Default	Description
spring.cloud.vault.app-role.role		Name of the role, optional, used for pull-mode.
spring.cloud.vault.app-role.role-id		The RoleId.
spring.cloud.vault.app-role.secret-id		The SecretId.
spring.cloud.vault.application-name	application	Application name for AppId authentication.
spring.cloud.vault.authentication		
spring.cloud.vault.aws-ec2.aws-ec2-path	aws-ec2	Mount path of the AWS-EC2 authentication backend.
spring.cloud.vault.aws-ec2.identity-document	169.254.169.254/latest/dynamic/instance-identity/pkcs7	URL of the AWS-EC2 PKCS7 identity document.
spring.cloud.vault.aws-ec2.nonce		Nonce used for AWS-EC2 authentication. An empty nonce defaults to nonce generation.
spring.cloud.vault.aws-ec2.role		Name of the role, optional.
spring.cloud.vault.aws-iam.aws-path	aws	Mount path of the AWS authentication backend.
spring.cloud.vault.aws-iam.endpoint-uri		STS server URI. @since 2.2
spring.cloud.vault.aws-iam.role		Name of the role, optional. Defaults to the friendly IAM name if not set.
spring.cloud.vault.aws-iam.server-name		Name of the server used to set {@code X-Vault-AWS-IAM-Server-ID} header in the headers of login requests.
spring.cloud.vault.azure-msi.azure-path	azure	Mount path of the Azure MSI authentication backend.
spring.cloud.vault.azure-msi.role		Name of the role.
spring.cloud.vault.config.lifecycle.enabled	true	Enable lifecycle management.
spring.cloud.vault.config.lifecycle.expiry-threshold		The expiry threshold. {@link Lease} is renewed the given {@link Duration} before it expires. @since 2.2

Name	Default	Description
spring.cloud.vault.config.lifecycle.lease-endpoints		Set the {@link LeaseEndpoints} to delegate renewal/revocation calls to. {@link LeaseEndpoints} encapsulates differences between Vault versions that affect the location of renewal/revocation endpoints. Can be {@link LeaseEndpoints#SysLeases} for version 0.8 or above of Vault or {@link LeaseEndpoints#Legacy} for older versions (the default). @since 2.2
spring.cloud.vault.config.lifecycle.min-renewal		The time period that is at least required before renewing a lease. @since 2.2
spring.cloud.vault.config.order	0	Used to set a {@link org.springframework.core.env.PropertySource} priority. This is useful to use Vault as an override on other property sources. @see org.springframework.core.PriorityOrdered
spring.cloud.vault.connection-timeout	5000	Connection timeout.
spring.cloud.vault.discovery.enabled	false	Flag to indicate that Vault server discovery is enabled (vault server URL will be looked up via discovery).
spring.cloud.vault.discovery.service-id	vault	Service id to locate Vault.
spring.cloud.vault.enabled	true	Enable Vault config server.
spring.cloud.vault.fail-fast	false	Fail fast if data cannot be obtained from Vault.
spring.cloud.vault.gcp-gce.gcp-path	gcp	Mount path of the Kubernetes authentication backend.
spring.cloud.vault.gcp-gce.role		Name of the role against which the login is being attempted.
spring.cloud.vault.gcp-gce.service-account		Optional service account id. Using the default id if left unconfigured.

Name	Default	Description
spring.cloud.vault.gcp-iam.credentials.encoded-key		The base64 encoded contents of an OAuth2 account private key in JSON format.
spring.cloud.vault.gcp-iam.credentials.location		Location of the OAuth2 credentials private key. <p> Since this is a Resource, the private key can be in a multitude of locations, such as a local file system, classpath, URL, etc.
spring.cloud.vault.gcp-iam.gcp-path	gcp	Mount path of the Kubernetes authentication backend.
spring.cloud.vault.gcp-iam.jwt-validity	15m	Validity of the JWT token.
spring.cloud.vault.gcp-iam.project-id		Overrides the GCP project Id.
spring.cloud.vault.gcp-iam.role		Name of the role against which the login is being attempted.
spring.cloud.vault.gcp-iam.service-account-id		Overrides the GCP service account Id.
spring.cloud.vault.generic.application-name	application	Application name to be used for the context.
spring.cloud.vault.generic.backend	secret	Name of the default backend.
spring.cloud.vault.generic.default-context	application	Name of the default context.
spring.cloud.vault.generic.enabled	true	Enable the generic backend.
spring.cloud.vault.generic.profile-separator	/	Profile-separator to combine application name and profile.
spring.cloud.vault.host	localhost	Vault server host.
spring.cloud.vault.kubernetes.kubernetes-path	kubernetes	Mount path of the Kubernetes authentication backend.
spring.cloud.vault.kubernetes.role		Name of the role against which the login is being attempted.
spring.cloud.vault.kubernetes.service-account-token-file	/var/run/secrets/kubernetes.io/serviceaccount/token	Path to the service account token file.
spring.cloud.vault.kv.application-name	application	Application name to be used for the context.

Name	Default	Description
spring.cloud.vault.kv.backend	secret	Name of the default backend.
spring.cloud.vault.kv.backend-version	2	Key-Value backend version. Currently supported versions are: - Version 1 (unversioned key-value backend). - Version 2 (versioned key-value backend).
spring.cloud.vault.kv.default-context	application	Name of the default context.
spring.cloud.vault.kv.enabled	false	Enable the kev-value backend.
spring.cloud.vault.kv.profile-separator	/	Profile-separator to combine application name and profile.
spring.cloud.vault.namespace		Vault namespace (requires Vault Enterprise).
spring.cloud.vault.pcf.instance-certificate		Path to the instance certificate (PEM). Defaults to {@code CF_INSTANCE_CERT} env variable.
spring.cloud.vault.pcf.instance-key		Path to the instance key (PEM). Defaults to {@code CF_INSTANCE_KEY} env variable.
spring.cloud.vault.pcf.pcf-path	pcf	Mount path of the Kubernetes authentication backend.
spring.cloud.vault.pcf.role		Name of the role against which the login is being attempted.
spring.cloud.vault.port	8200	Vault server port.
spring.cloud.vault.read-timeout	15000	Read timeout.
spring.cloud.vault.scheme	https	Protocol scheme. Can be either "http" or "https".
spring.cloud.vault.ssl.cert-auth-path	cert	Mount path of the TLS cert authentication backend.
spring.cloud.vault.ssl.key-store		Trust store that holds certificates and private keys.
spring.cloud.vault.ssl.key-store-password		Password used to access the key store.
spring.cloud.vault.ssl.trust-store		Trust store that holds SSL certificates.

Name	Default	Description
spring.cloud.vault.ssl.trust-store-password		Password used to access the trust store.
spring.cloud.vault.token		Static vault token. Required if {@link #authentication} is {@code TOKEN}.
spring.cloud.vault.uri		Vault URI. Can be set with scheme, host and port.
spring.cloud.zookeeper.base-sleep-time-ms	50	Initial amount of time to wait between retries.
spring.cloud.zookeeper.block-until-connected-unit		The unit of time related to blocking on connection to Zookeeper.
spring.cloud.zookeeper.block-until-connected-wait	10	Wait time to block on connection to Zookeeper.
spring.cloud.zookeeper.connect-string	localhost:2181	Connection string to the Zookeeper cluster.
spring.cloud.zookeeper.connection-timeout		The configured connection timeout in milliseconds.
spring.cloud.zookeeper.default-health-endpoint		Default health endpoint that will be checked to verify that a dependency is alive.
spring.cloud.zookeeper.dependencies		Mapping of alias to ZookeeperDependency. From Ribbon perspective the alias is actually serviceID since Ribbon can't accept nested structures in serviceID.
spring.cloud.zookeeper.dependency-configurations		
spring.cloud.zookeeper.dependency-names		
spring.cloud.zookeeper.discovery.enabled	true	
spring.cloud.zookeeper.discovery.initial-status		The initial status of this instance (defaults to {@link StatusConstants#STATUS_UP}).

Name	Default	Description
spring.cloud.zookeeper.discovery.instance-host		Predefined host with which a service can register itself in Zookeeper. Corresponds to the {code address} from the URI spec.
spring.cloud.zookeeper.discovery.instance-id		Id used to register with zookeeper. Defaults to a random UUID.
spring.cloud.zookeeper.discovery.instance-port		Port to register the service under (defaults to listening port).
spring.cloud.zookeeper.discovery.instance-ssl-port		Ssl port of the registered service.
spring.cloud.zookeeper.discovery.metadata		Gets the metadata name/value pairs associated with this instance. This information is sent to zookeeper and can be used by other instances.
spring.cloud.zookeeper.discovery.order	0	Order of the discovery client used by `CompositeDiscoveryClient` for sorting available clients.
spring.cloud.zookeeper.discovery.register	true	Register as a service in zookeeper.
spring.cloud.zookeeper.discovery.root	/services	Root Zookeeper folder in which all instances are registered.
spring.cloud.zookeeper.discovery.uri-spec	{scheme}://{address}:{port}	The URI specification to resolve during service registration in Zookeeper.
spring.cloud.zookeeper.enabled	true	Is Zookeeper enabled.
spring.cloud.zookeeper.max-retries	10	Max number of times to retry.
spring.cloud.zookeeper.max-sleep-ms	500	Max time in ms to sleep on each retry.
spring.cloud.zookeeper.prefix		Common prefix that will be applied to all Zookeeper dependencies' paths.

Name	Default	Description
spring.cloud.zookeeper.session-timeout		The configured/negotiated session timeout in milliseconds. Please refer to cwiki.apache.org/confluence/display/CURATOR/TN14'>Curator's Tech Note 14 to understand how Curator implements connection sessions. @see cwiki.apache.org/confluence/display/CURATOR/TN14'>Curator's Tech Note 14
spring.sleuth.annotation.enabled	true	
spring.sleuth.async.configurer.enabled	true	Enable default AsyncConfigurer.
spring.sleuth.async.enabled	true	Enable instrumenting async related components so that the tracing information is passed between threads.
spring.sleuth.async.ignored-beans		List of {@link java.util.concurrent.Executor} bean names that should be ignored and not wrapped in a trace representation.

Name	Default	Description
spring.sleuth.baggage-keys		List of baggage key names that should be propagated out of process. These keys will be prefixed with `baggage` before the actual key. This property is set in order to be backward compatible with previous Sleuth versions. @see brave.propagation.ExtraFieldPropagation.FactoryBuilder#addPrefixedFields(String, java.util.Collection)
spring.sleuth.baggage.correlation-enabled	true	Adds a {@link CorrelationScopeDecorator} to put baggage values into the correlation context.
spring.sleuth.baggage.correlation-fields		A list of {@link BaggageField#name() fields} to add to correlation (MDC) context. @see CorrelationScopeConfig.SingleCorrelationField#create(BaggageField)
spring.sleuth.baggage.local-fields		Same as {@link #remoteFields} except that this field is not propagated to remote services. @see BaggagePropagationConfig.SingleBaggageField#local(BaggageField)
spring.sleuth.baggage.remote-fields		List of fields that are referenced the same in-process as it is on the wire. For example, the field "x-vcap-request-id" would be set as-is including the prefix. @see BaggagePropagationConfig.SingleBaggageField#remote(BaggageField) @see BaggagePropagationConfig.SingleBaggageField.Builder#addKeyName(String)

Name	Default	Description
spring.sleuth.baggage.tag-fields		A list of {@link BaggageField#name() fields} to tag into the span. @see Tags#BAGGAGE_FIELD
spring.sleuth.circuitbreaker.enabled	true	Enable Spring Cloud CircuitBreaker instrumentation.
spring.sleuth.enabled	true	
spring.sleuth.feign.enabled	true	Enable span information propagation when using Feign.
spring.sleuth.feign.processor.enabled	true	Enable post processor that wraps Feign Context in its tracing representations.
spring.sleuth.grpc.enabled	true	Enable span information propagation when using GRPC.
spring.sleuth.http.enabled	true	
spring.sleuth.http.legacy.enabled	false	Enables the legacy Sleuth setup.
spring.sleuth.hystrix.strategy.enabled	true	Enable custom HystrixConcurrencyStrategy that wraps all Callable instances into their Sleuth representative - the TraceCallable.
spring.sleuth.hystrix.strategy.passthrough	false	When enabled the tracing information is passed to the Hystrix execution threads but spans are not created for each execution.
spring.sleuth.integration.enabled	true	Enable Spring Integration sleuth instrumentation.
spring.sleuth.integration.patterns	[!hystrixStreamOutput*, , **!channel**]	An array of patterns against which channel names will be matched. @see org.springframework.integration.config.GlobalChannelInterceptor#patterns() Defaults to any channel name not matching the Hystrix Stream and functional Stream channel names.
spring.sleuth.integration.websockets.enabled	true	Enable tracing for WebSockets.

Name	Default	Description
spring.sleuth.keys.http.headers		Additional headers that should be added as tags if they exist. If the header value is multi-valued, the tag value will be a comma-separated, single-quoted list.
spring.sleuth.keys.http.prefix	http.	Prefix for header names if they are added as tags.
spring.sleuth.local-keys		Same as {@link #propagationKeys} except that this field is not propagated to remote services. @see brave.propagation.ExtraFieldPropagation.FactoryBuilder#addRedactedField(String) @deprecated use {@code spring.sleuth.baggage.local-fields} property
spring.sleuth.log.slf4j.enabled	true	Enable a {@link Slf4jScopeDecorator} that prints tracing information in the logs.
spring.sleuth.log.slf4j.whitelisted-mdc-keys		A list of keys to be put from baggage to MDC. @deprecated use spring.sleuth.baggage.correlation-fields property
spring.sleuth.messaging.enabled	false	Should messaging be turned on.
spring.sleuth.messaging.jms.enabled	true	Enable tracing of JMS.
spring.sleuth.messaging.jms.remote-service-name	jms	
spring.sleuth.messaging.kafka.enabled	true	Enable tracing of Kafka.
spring.sleuth.messaging.kafka.mapper.enabled	true	Enable DefaultKafkaHeaderMapper tracing for Kafka.
spring.sleuth.messaging.kafka.remote-service-name	kafka	
spring.sleuth.messaging.rabbit.enabled	true	Enable tracing of RabbitMQ.

Name	Default	Description
spring.sleuth.messaging.rabbit.remote-service-name	rabbitmq	
spring.sleuth.opentracing.enabled	true	
spring.sleuth.propagation-keys		List of fields that are referenced the same in-process as it is on the wire. For example, the name "x-vcap-request-id" would be set as-is including the prefix. <p> Note: {@code fieldName} will be implicitly lower-cased. @see brave.propagation.ExtraFieldPropagation.FactoryBuilder#addField(String) @deprecated use {@code spring.sleuth.baggage.remote-fields} property
spring.sleuth.propagation.tag.enabled	true	Enables a {@link TagPropagationFinishedSpanHandler} that adds extra propagated fields to span tags.
spring.sleuth.propagation.tag.whitelisted-keys		A list of keys to be put from extra propagation fields to span tags.
spring.sleuth.reactor.decorate-on-each	true	When true decorates on each operator, will be less performing, but logging will always contain the tracing entries in each operator. When false decorates on last operator, will be more performing, but logging might not always contain the tracing entries.
spring.sleuth.reactor.enabled	true	When true enables instrumentation for reactor.
spring.sleuth.redis.enabled	true	Enable span information propagation when using Redis.
spring.sleuth.redis.remote-service-name	redis	Service name for the remote Redis endpoint.
spring.sleuth.rpc.enabled	true	Enable tracing of RPC.

Name	Default	Description
spring.sleuth.rxjava.schedulers.hook.enabled	true	Enable support for RxJava via RxJavaSchedulersHook.
spring.sleuth.rxjava.schedulers.ignoredthreads	[HystrixMetricPoller, ^RxComputation.*$]	Thread names for which spans will not be sampled.
spring.sleuth.sampler.probability		Probability of requests that should be sampled. E.g. 1.0 - 100% requests should be sampled. The precision is whole-numbers only (i.e. there's no support for 0.1% of the traces).
spring.sleuth.sampler.rate	10	A rate per second can be a nice choice for low-traffic endpoints as it allows you surge protection. For example, you may never expect the endpoint to get more than 50 requests per second. If there was a sudden surge of traffic, to 5000 requests per second, you would still end up with 50 traces per second. Conversely, if you had a percentage, like 10%, the same surge would end up with 500 traces per second, possibly overloading your storage. Amazon X-Ray includes a rate-limited sampler (named Reservoir) for this purpose. Brave has taken the same approach via the {@link brave.sampler.RateLimitingSampler}.
spring.sleuth.scheduled.enabled	true	Enable tracing for {@link org.springframework.scheduling.annotation.Scheduled}.
spring.sleuth.scheduled.skip-pattern	org.springframework.cloud.netflix.hystrix.stream.HystrixStreamTask	Pattern for the fully qualified name of a class that should be skipped.
spring.sleuth.supports-join	true	True means the tracing system supports sharing a span ID between a client and server.

Name	Default	Description
spring.sleuth.trace-id128	false	When true, generate 128-bit trace IDs instead of 64-bit ones.
spring.sleuth.web.additional-skip-pattern		Additional pattern for URLs that should be skipped in tracing. This will be appended to the {@link SleuthWebProperties#skipPattern}.
spring.sleuth.web.client.enabled	true	Enable interceptor injecting into {@link org.springframework.web.client.RestTemplate}.
spring.sleuth.web.client.skip-pattern		Pattern for URLs that should be skipped in client side tracing.
spring.sleuth.web.enabled	true	When true enables instrumentation for web applications.
spring.sleuth.web.exception-logging-filter-enabled	true	Flag to toggle the presence of a filter that logs thrown exceptions.
spring.sleuth.web.exception-throwing-filter-enabled	true	Flag to toggle the presence of a filter that logs thrown exceptions. @deprecated use {@link #exceptionLoggingFilterEnabled}
spring.sleuth.web.filter-order		Order in which the tracing filters should be registered. Defaults to {@link TraceHttpAutoConfiguration#TRACING_FILTER_ORDER}.
spring.sleuth.web.ignore-auto-configured-skip-patterns	false	If set to true, auto-configured skip patterns will be ignored. @see TraceWebAutoConfiguration
spring.sleuth.web.skip-pattern	/api-docs.*\	/swagger.*\
spring.sleuth.zuul.enabled	true	Enable span information propagation when using Zuul.
spring.zipkin.activemq.message-max-bytes	100000	Maximum number of bytes for a given message with spans sent to Zipkin over ActiveMQ.

Name	Default	Description
spring.zipkin.activemq.queue	zipkin	Name of the ActiveMQ queue where spans should be sent to Zipkin.
spring.zipkin.base-url	localhost:9411/	URL of the zipkin query server instance. You can also provide the service id of the Zipkin server if Zipkin's registered in service discovery (e.g. zipkinserver/).
spring.zipkin.compression.enabled	false	
spring.zipkin.discovery-client-enabled		If set to {@code false}, will treat the {@link ZipkinProperties#baseUrl} as a URL always.
spring.zipkin.enabled	true	Enables sending spans to Zipkin.
spring.zipkin.encoder		Encoding type of spans sent to Zipkin. Set to {@link SpanBytesEncoder#JSON_V1} if your server is not recent.
spring.zipkin.kafka.topic	zipkin	Name of the Kafka topic where spans should be sent to Zipkin.
spring.zipkin.locator.discovery.enabled	false	Enabling of locating the host name via service discovery.
spring.zipkin.message-timeout	1	Timeout in seconds before pending spans will be sent in batches to Zipkin.
spring.zipkin.rabbitmq.addresses		Addresses of the RabbitMQ brokers used to send spans to Zipkin
spring.zipkin.rabbitmq.queue	zipkin	Name of the RabbitMQ queue where spans should be sent to Zipkin.
spring.zipkin.sender.type		Means of sending spans to Zipkin.
spring.zipkin.service.name		The name of the service, from which the Span was sent via HTTP, that should appear in Zipkin.

Name	Default	Description
stubrunner.amqp.enabled	false	Whether to enable support for Stub Runner and AMQP.
stubrunner.amqp.mockCOnnection	true	Whether to enable support for Stub Runner and AMQP mocked connection factory.
stubrunner.classifier	stubs	The classifier to use by default in ivy co-ordinates for a stub.
stubrunner.cloud.consul.enabled	true	Whether to enable stubs registration in Consul.
stubrunner.cloud.delegate.enabled	true	Whether to enable DiscoveryClient's Stub Runner implementation.
stubrunner.cloud.enabled	true	Whether to enable Spring Cloud support for Stub Runner.
stubrunner.cloud.eureka.enabled	true	Whether to enable stubs registration in Eureka.
stubrunner.cloud.loadbalancer.enabled	true	Whether to enable Stub Runner's Spring Cloud Load Balancer integration.
stubrunner.cloud.stubbed.discovery.enabled	true	Whether Service Discovery should be stubbed for Stub Runner. If set to false, stubs will get registered in real service discovery.
stubrunner.cloud.zookeeper.enabled	true	Whether to enable stubs registration in Zookeeper.
stubrunner.consumer-name		You can override the default {@code spring.application.name} of this field by setting a value to this parameter.
stubrunner.delete-stubs-after-test	true	If set to {@code false} will NOT delete stubs from a temporary folder after running tests.
stubrunner.fail-on-no-stubs	true	When enabled, this flag will tell stub runner to throw an exception when no stubs / contracts were found.

Name	Default	Description
stubrunner.generate-stubs	false	When enabled, this flag will tell stub runner to not load the generated stubs, but convert the found contracts at runtime to a stub format and run those stubs.
stubrunner.http-server-stub-configurer		Configuration for an HTTP server stub.
stubrunner.ids	[]	The ids of the stubs to run in "ivy" notation ([groupId]:artifactId:[version]:[classifier][:port]). {@code groupId}, {@code classifier}, {@code version} and {@code port} can be optional.
stubrunner.ids-to-service-ids		Mapping of Ivy notation based ids to serviceIds inside your application. Example "a:b" -> "myService" "artifactId" -> "myOtherService"
stubrunner.integration.enabled	true	Whether to enable Stub Runner integration with Spring Integration.
stubrunner.jms.enabled	true	Whether to enable Stub Runner integration with Spring JMS.
stubrunner.kafka.enabled	true	Whether to enable Stub Runner integration with Spring Kafka.
stubrunner.kafka.initializer.enabled	true	Whether to allow Stub Runner to take care of polling for messages instead of the KafkaStubMessages component. The latter should be used only on the producer side.
stubrunner.mappings-output-folder		Dumps the mappings of each HTTP server to the selected folder.
stubrunner.max-port	15000	Max value of a port for the automatically started WireMock server.
stubrunner.min-port	10000	Min value of a port for the automatically started WireMock server.

Name	Default	Description
stubrunner.password		Repository password.
stubrunner.properties		Map of properties that can be passed to custom {@link org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder}.
stubrunner.proxy-host		Repository proxy host.
stubrunner.proxy-port		Repository proxy port.
stubrunner.stream.enabled	true	Whether to enable Stub Runner integration with Spring Cloud Stream.
stubrunner.stubs-mode		Pick where the stubs should come from.
stubrunner.stubs-per-consumer	false	Should only stubs for this particular consumer get registered in HTTP server stub.
stubrunner.username		Repository username.
wiremock.placeholders.enabled	true	Flag to indicate that http URLs in generated wiremock stubs should be filtered to add or resolve a placeholder for a dynamic port.
wiremock.reset-mappings-after-each-test	false	
wiremock.rest-template-ssl-enabled	false	
wiremock.server.files	[]	
wiremock.server.https-port	-1	
wiremock.server.https-port-dynamic	false	
wiremock.server.port	8080	
wiremock.server.port-dynamic	false	
wiremock.server.stubs	[]	
zuul.ribbon-isolation-strategy		
zuul.ribbon.eager-load.enabled	false	Enables eager loading of Ribbon clients on startup.