

Spring Boot Reference Documentation

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

Version 3.2.7

Table of Contents

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

Maven	14
Gradle	15
4.4.5. Writing the Code	15
The <code>@RestController</code> and <code>@RequestMapping</code> Annotations	17
The <code>@SpringBootApplication</code> Annotation	17
The “main” Method	17
4.4.6. Running the Example	17
Maven	17
Gradle	18
4.4.7. Creating an Executable Jar	19
Maven	19
Gradle	20
4.5. What to Read Next	21
5. Upgrading Spring Boot	22
5.1. Upgrading From 1.x	22
5.2. Upgrading to a New Feature Release	22
5.3. Upgrading the Spring Boot CLI	22
5.4. What to Read Next	22
6. Developing with Spring Boot	24
6.1. Build Systems	24
6.1.1. Dependency Management	24
6.1.2. Maven	24
6.1.3. Gradle	24
6.1.4. Ant	25
6.1.5. Starters	26
6.2. Structuring Your Code	30
6.2.1. Using the “default” Package	30
6.2.2. Locating the Main Application Class	31
6.3. Configuration Classes	32
6.3.1. Importing Additional Configuration Classes	32
6.3.2. Importing XML Configuration	32
6.4. Auto-configuration	32
6.4.1. Gradually Replacing Auto-configuration	33
6.4.2. Disabling Specific Auto-configuration Classes	33
6.4.3. Auto-configuration Packages	34
6.5. Spring Beans and Dependency Injection	34
6.6. Using the <code>@SpringBootApplication</code> Annotation	36
6.7. Running Your Application	39
6.7.1. Running From an IDE	39
6.7.2. Running as a Packaged Application	39
6.7.3. Using the Maven Plugin	39

6.7.4. Using the Gradle Plugin	40
6.7.5. Hot Swapping	40
6.8. Developer Tools	40
6.8.1. Diagnosing Classloading Issues	41
6.8.2. Property Defaults	41
6.8.3. Automatic Restart	42
Logging Changes in Condition Evaluation	44
Excluding Resources	44
Watching Additional Paths	45
Disabling Restart	45
Using a Trigger File	46
Customizing the Restart Classloader	47
Known Limitations	48
6.8.4. LiveReload	48
6.8.5. Global Settings	48
Configuring File System Watcher	49
6.8.6. Remote Applications	50
Running the Remote Client Application	50
Remote Update	52
6.9. Packaging Your Application for Production	52
6.10. What to Read Next	52
7. Core Features	53
7.1. SpringApplication	53
7.1.1. Startup Failure	54
7.1.2. Lazy Initialization	55
7.1.3. Customizing the Banner	56
7.1.4. Customizing SpringApplication	57
7.1.5. Fluent Builder API	58
7.1.6. Application Availability	59
Liveness State	59
Readiness State	59
Managing the Application Availability State	59
7.1.7. Application Events and Listeners	63
7.1.8. Web Environment	64
7.1.9. Accessing Application Arguments	64
7.1.10. Using the ApplicationRunner or CommandLineRunner	65
7.1.11. Application Exit	66
7.1.12. Admin Features	68
7.1.13. Application Startup tracking	68
Virtual threads	69
7.2. Externalized Configuration	70

7.2.1. Accessing Command Line Properties	72
7.2.2. JSON Application Properties	72
7.2.3. External Application Properties	73
Optional Locations	75
Wildcard Locations	75
Profile Specific Files	76
Importing Additional Data	77
Importing Extensionless Files	78
Using Configuration Trees	79
Property Placeholders	81
Working With Multi-Document Files	82
Activation Properties	83
7.2.4. Encrypting Properties	84
7.2.5. Working With YAML	84
Mapping YAML to Properties	84
Directly Loading YAML	85
7.2.6. Configuring Random Values	85
7.2.7. Configuring System Environment Properties	86
7.2.8. Type-safe Configuration Properties	86
JavaBean Properties Binding	86
Constructor Binding	89
Enabling @ConfigurationProperties-annotated Types	92
Using @ConfigurationProperties-annotated Types	94
Third-party Configuration	96
Relaxed Binding	97
Merging Complex Types	100
Properties Conversion	104
@ConfigurationProperties Validation	111
@ConfigurationProperties vs. @Value	114
7.3. Profiles	115
7.3.1. Adding Active Profiles	117
7.3.2. Profile Groups	118
7.3.3. Programmatically Setting Profiles	119
7.3.4. Profile-specific Configuration Files	119
7.4. Logging	119
7.4.1. Log Format	119
7.4.2. Console Output	121
Color-coded Output	121
7.4.3. File Output	122
7.4.4. File Rotation	122
7.4.5. Log Levels	123

7.4.6. Log Groups	123
7.4.7. Using a Log Shutdown Hook	124
7.4.8. Custom Log Configuration	125
7.4.9. Logback Extensions	127
Profile-specific Configuration	128
Environment Properties	128
7.4.10. Log4j2 Extensions	128
Profile-specific Configuration	129
Environment Properties Lookup	129
Log4j2 System Properties	130
7.5. Internationalization	130
7.6. Aspect-Oriented Programming	131
7.7. JSON	131
7.7.1. Jackson	131
Custom Serializers and Deserializers	131
Mixins	135
7.7.2. Gson	135
7.7.3. JSON-B	136
7.8. Task Execution and Scheduling	136
7.9. Testing	138
7.9.1. Test Scope Dependencies	138
7.9.2. Testing Spring Applications	139
7.9.3. Testing Spring Boot Applications	139
Detecting Web Application Type	140
Detecting Test Configuration	140
Using the Test Configuration Main Method	141
Excluding Test Configuration	144
Using Application Arguments	145
Testing With a Mock Environment	146
Testing With a Running Server	150
Customizing WebTestClient	152
Using JMX	153
Using Observations	154
Using Metrics	154
Using Tracing	154
Mocking and Spying Beans	155
Auto-configured Tests	158
Auto-configured JSON Tests	159
Auto-configured Spring MVC Tests	162
Auto-configured Spring WebFlux Tests	166
Auto-configured Spring GraphQL Tests	169

Auto-configured Data Cassandra Tests	174
Auto-configured Data Couchbase Tests	175
Auto-configured Data Elasticsearch Tests	176
Auto-configured Data JPA Tests	177
Auto-configured JDBC Tests	180
Auto-configured Data JDBC Tests	181
Auto-configured Data R2DBC Tests	181
Auto-configured jOOQ Tests	182
Auto-configured Data MongoDB Tests	183
Auto-configured Data Neo4j Tests	184
Auto-configured Data Redis Tests	185
Auto-configured Data LDAP Tests	186
Auto-configured REST Clients	188
Auto-configured Spring REST Docs Tests	192
Auto-configured Spring Web Services Tests	203
Additional Auto-configuration and Slicing	207
User Configuration and Slicing	208
Using Spock to Test Spring Boot Applications	212
7.9.4. Testcontainers	212
Service Connections	214
Dynamic Properties	217
7.9.5. Test Utilities	219
ConfigDataApplicationContextInitializer	219
TestPropertyValues	220
OutputCapture	221
TestRestTemplate	222
7.10. Docker Compose Support	226
7.10.1. Prerequisites	227
7.10.2. Service Connections	227
7.10.3. Custom Images	228
7.10.4. Skipping Specific Containers	229
7.10.5. Using a Specific Compose File	229
7.10.6. Waiting for Container Readiness	230
7.10.7. Controlling the Docker Compose Lifecycle	230
7.10.8. Activating Docker Compose Profiles	231
7.10.9. Using Docker Compose in Tests	232
7.11. Testcontainers Support	232
7.11.1. Using Testcontainers at Development Time	232
Contributing Dynamic Properties at Development Time	235
Importing Testcontainer Declaration Classes	236
Using DevTools with Testcontainers at Development Time	238

7.12. Creating Your Own Auto-configuration	239
7.12.1. Understanding Auto-configured Beans.....	239
7.12.2. Locating Auto-configuration Candidates	239
7.12.3. Condition Annotations	240
Class Conditions	240
Bean Conditions	242
Property Conditions	244
Resource Conditions.....	244
Web Application Conditions.....	244
SpEL Expression Conditions.....	244
7.12.4. Testing your Auto-configuration.....	244
Simulating a Web Context.....	248
Overriding the Classpath	248
7.12.5. Creating Your Own Starter.....	249
Naming	249
Configuration keys	249
The “autoconfigure” Module	251
Starter Module	252
7.13. Kotlin Support.....	253
7.13.1. Requirements	253
7.13.2. Null-safety	253
7.13.3. Kotlin API	254
runApplication	254
Extensions.....	254
7.13.4. Dependency management	255
7.13.5. @ConfigurationProperties	255
7.13.6. Testing	255
7.13.7. Resources	256
Further reading	256
Examples.....	256
7.14. SSL	256
7.14.1. Configuring SSL With Java KeyStore Files	256
7.14.2. Configuring SSL With PEM-encoded Certificates	257
7.14.3. Applying SSL Bundles	259
7.14.4. Using SSL Bundles	260
7.14.5. Reloading SSL bundles	261
7.15. What to Read Next	262
8. Web	263
8.1. Servlet Web Applications	263
8.1.1. The “Spring Web MVC Framework”.....	263
Spring MVC Auto-configuration	269

Spring MVC Conversion Service	269
HttpMessageConverters	270
MessageCodesResolver	271
Static Content	271
Welcome Page	274
Custom Favicon	274
Path Matching and Content Negotiation	274
ConfigurableWebBindingInitializer	276
Template Engines	276
Error Handling	277
CORS Support	285
8.1.2. JAX-RS and Jersey	286
8.1.3. Embedded Servlet Container Support	288
Servlets, Filters, and Listeners	288
Servlet Context Initialization	288
The ServletWebServerApplicationContext	289
Customizing Embedded Servlet Containers	290
JSP Limitations	295
8.2. Reactive Web Applications	295
8.2.1. The “Spring WebFlux Framework”	295
Spring WebFlux Auto-configuration	301
Spring WebFlux Conversion Service	301
HTTP Codecs with HttpMessageReaders and HttpMessageWriters	302
Static Content	303
Welcome Page	304
Template Engines	304
Error Handling	304
Web Filters	308
8.2.2. Embedded Reactive Server Support	309
Customizing Reactive Servers	309
8.2.3. Reactive Server Resources Configuration	312
8.3. Graceful Shutdown	312
8.4. Spring Security	313
8.4.1. MVC Security	314
8.4.2. WebFlux Security	314
8.4.3. OAuth2	316
Client	316
Resource Server	322
Authorization Server	324
8.4.4. SAML 2.0	327
Relying Party	327

8.5. Spring Session	330
8.6. Spring for GraphQL	331
8.6.1. GraphQL Schema	332
8.6.2. GraphQL RuntimeWiring	333
8.6.3. Querydsl and QueryByExample Repositories Support	334
8.6.4. Transports	335
HTTP and WebSocket	335
RSocket	335
8.6.5. Exception Handling	337
8.6.6. GraphiQL and Schema printer	337
8.7. Spring HATEOAS	337
8.8. What to Read Next	338
9. Data	339
9.1. SQL Databases	339
9.1.1. Configure a DataSource	339
Embedded Database Support	339
Connection to a Production Database	340
DataSource Configuration	340
Supported Connection Pools	341
Connection to a JNDI DataSource	342
9.1.2. Using JdbcTemplate	342
9.1.3. Using JdbcClient	344
9.1.4. JPA and Spring Data JPA	345
Entity Classes	345
Spring Data JPA Repositories	347
Spring Data Envers Repositories	349
Creating and Dropping JPA Databases	349
Open EntityManager in View	350
9.1.5. Spring Data JDBC	350
9.1.6. Using H2's Web Console	351
Changing the H2 Console's Path	351
Accessing the H2 Console in a Secured Application	351
9.1.7. Using jOOQ	353
Code Generation	353
Using DSLContext	354
jOOQ SQL Dialect	356
Customizing jOOQ	356
9.1.8. Using R2DBC	356
Embedded Database Support	359
Using DatabaseClient	360
Spring Data R2DBC Repositories	360

9.2. Working with NoSQL Technologies	361
9.2.1. Redis	362
Connecting to Redis	362
9.2.2. MongoDB	364
Connecting to a MongoDB Database	364
MongoTemplate	368
Spring Data MongoDB Repositories	369
9.2.3. Neo4j	370
Connecting to a Neo4j Database	370
Spring Data Neo4j Repositories	372
9.2.4. Elasticsearch	375
Connecting to Elasticsearch Using REST clients	375
Connecting to Elasticsearch by Using Spring Data	376
Spring Data Elasticsearch Repositories	377
9.2.5. Cassandra	378
Connecting to Cassandra	378
Spring Data Cassandra Repositories	381
9.2.6. Couchbase	382
Connecting to Couchbase	382
Spring Data Couchbase Repositories	383
9.2.7. LDAP	385
Connecting to an LDAP Server	385
Spring Data LDAP Repositories	386
Embedded In-memory LDAP Server	387
9.2.8. InfluxDB	388
Connecting to InfluxDB	388
9.3. What to Read Next	388
10. Messaging	389
10.1. JMS	389
10.1.1. ActiveMQ "Classic" Support	389
10.1.2. ActiveMQ Artemis Support	390
10.1.3. Using a JNDI ConnectionFactory	392
10.1.4. Sending a Message	392
10.1.5. Receiving a Message	393
10.2. AMQP	397
10.2.1. RabbitMQ Support	397
10.2.2. Sending a Message	398
10.2.3. Sending a Message To A Stream	400
10.2.4. Receiving a Message	401
10.3. Apache Kafka Support	404
10.3.1. Sending a Message	405

10.3.2. Receiving a Message	406
10.3.3. Kafka Streams	407
10.3.4. Additional Kafka Properties	408
10.3.5. Testing with Embedded Kafka	410
10.4. Apache Pulsar Support	412
10.4.1. Connecting to Pulsar	412
Authentication	412
SSL	413
10.4.2. Connecting to Pulsar Reactively	413
10.4.3. Connecting to Pulsar Administration	414
Authentication	414
10.4.4. Sending a Message	414
10.4.5. Sending a Message Reactively	416
10.4.6. Receiving a Message	417
10.4.7. Receiving a Message Reactively	418
10.4.8. Reading a Message	419
10.4.9. Reading a Message Reactively	420
10.4.10. Additional Pulsar Properties	421
10.5. RSocket	422
10.5.1. RSocket Strategies Auto-configuration	422
10.5.2. RSocket server Auto-configuration	422
10.5.3. Spring Messaging RSocket support	423
10.5.4. Calling RSocket Services with RSocketRequester	423
10.6. Spring Integration	424
10.7. WebSockets	426
10.8. What to Read Next	426
11. IO	427
11.1. Caching	427
11.1.1. Supported Cache Providers	428
Generic	430
JCache (JSR-107)	430
Hazelcast	431
Infinispan	431
Couchbase	432
Redis	434
Caffeine	436
Cache2k	437
Simple	438
None	438
11.1.2. Hazelcast	439
11.1.3. Quartz Scheduler	440

11.4. Sending Email	443
11.5. Validation	444
11.6. Calling REST Services	445
11.6.1. WebClient	446
WebClient Runtime	447
WebClient Customization	448
WebClient SSL Support	448
11.6.2. RestClient	450
RestClient Customization	451
RestClient SSL Support	452
11.6.3. RestTemplate	455
RestTemplate Customization	457
RestTemplate SSL Support	461
11.6.4. HTTP Client Detection for RestClient and RestTemplate	462
11.7. Web Services	462
11.7.1. Calling Web Services with WebServiceTemplate	463
11.8. Distributed Transactions With JTA	466
11.8.1. Using a Jakarta EE Managed Transaction Manager	466
11.8.2. Mixing XA and Non-XA JMS Connections	466
11.8.3. Supporting an Embedded Transaction Manager	467
11.9. What to Read Next	467
12. Container Images	468
12.1. Efficient Container Images	468
12.1.1. Layering Docker Images	468
12.2. Dockerfiles	469
12.3. Cloud Native Buildpacks	470
12.4. What to Read Next	471
13. Production-ready Features	472
13.1. Enabling Production-ready Features	472
13.2. Endpoints	472
13.2.1. Enabling Endpoints	474
13.2.2. Exposing Endpoints	475
13.2.3. Security	476
Cross Site Request Forgery Protection	479
13.2.4. Configuring Endpoints	479
13.2.5. Sanitize Sensitive Values	480
13.2.6. Hypermedia for Actuator Web Endpoints	481
13.2.7. CORS Support	481
13.2.8. Implementing Custom Endpoints	482
Receiving Input	483
Custom Web Endpoints	484

Servlet Endpoints	485
Controller Endpoints	485
13.2.9. Health Information	486
Auto-configured HealthIndicators	486
Writing Custom HealthIndicators	487
Reactive Health Indicators	490
Auto-configured ReactiveHealthIndicators	492
Health Groups	492
DataSource Health	494
13.2.10. Kubernetes Probes	494
Checking External State With Kubernetes Probes	496
Application Lifecycle and Probe States	497
13.2.11. Application Information	497
Auto-configured InfoContributors	498
Custom Application Information	498
Git Commit Information	499
Build Information	500
Java Information	500
OS Information	500
Writing Custom InfoContributors	500
13.3. Monitoring and Management Over HTTP	501
13.3.1. Customizing the Management Endpoint Paths	502
13.3.2. Customizing the Management Server Port	503
13.3.3. Configuring Management-specific SSL	503
13.3.4. Customizing the Management Server Address	505
13.3.5. Disabling HTTP Endpoints	505
13.4. Monitoring and Management over JMX	506
13.4.1. Customizing MBean Names	506
13.4.2. Disabling JMX Endpoints	507
13.5. Observability	507
13.5.1. Common tags	509
13.5.2. Preventing Observations	509
13.5.3. OpenTelemetry Support	510
13.5.4. Micrometer Observation Annotations support	510
13.6. Loggers	510
13.6.1. Configure a Logger	511
13.7. Metrics	511
13.7.1. Getting started	512
13.7.2. Supported Monitoring Systems	515
AppOptics	515
Atlas	516

Datadog	516
Dynatrace	517
Elastic	520
Ganglia	521
Graphite	521
Humio	523
Influx	524
JMX	524
KairosDB	526
New Relic	526
OpenTelemetry	528
Prometheus	528
SignalFx	529
Simple	529
Stackdriver	530
StatsD	530
Wavefront	531
13.7.3. Supported Metrics and Meters	532
JVM Metrics	532
System Metrics	533
Application Startup Metrics	533
Logger Metrics	533
Task Execution and Scheduling Metrics	533
JMS Metrics	533
Spring MVC Metrics	533
Spring WebFlux Metrics	534
Jersey Server Metrics	534
HTTP Client Metrics	535
Tomcat Metrics	535
Cache Metrics	535
Spring Batch Metrics	536
Spring GraphQL Metrics	536
DataSource Metrics	536
Hibernate Metrics	536
Spring Data Repository Metrics	537
RabbitMQ Metrics	537
Spring Integration Metrics	537
Kafka Metrics	538
MongoDB Metrics	538
Jetty Metrics	541
@Timed Annotation Support	541

Redis Metrics	541
13.7.4. Registering Custom Metrics	541
13.7.5. Customizing Individual Metrics	543
Common Tags	544
Per-meter Properties	545
13.7.6. Metrics Endpoint	546
13.7.7. Integration with Micrometer Observation	547
13.8. Tracing	547
13.8.1. Supported Tracers	547
13.8.2. Getting Started	547
13.8.3. Logging Correlation IDs	549
13.8.4. Propagating Traces	550
13.8.5. Tracer Implementations	550
OpenTelemetry With Zipkin	550
OpenTelemetry With Wavefront	550
OpenTelemetry With OTLP	551
OpenZipkin Brave With Zipkin	551
OpenZipkin Brave With Wavefront	551
13.8.6. Integration with Micrometer Observation	551
13.8.7. Creating Custom Spans	551
13.8.8. Baggage	552
13.8.9. Tests	553
13.9. Auditing	553
13.9.1. Custom Auditing	554
13.10. Recording HTTP Exchanges	554
13.10.1. Custom HTTP Exchange Recording	554
13.11. Process Monitoring	554
13.11.1. Extending Configuration	555
13.11.2. Programmatically Enabling Process Monitoring	555
13.12. Cloud Foundry Support	555
13.12.1. Disabling Extended Cloud Foundry Actuator Support	555
13.12.2. Cloud Foundry Self-signed Certificates	556
13.12.3. Custom Context Path	556
13.13. What to Read Next	561
14. Deploying Spring Boot Applications	562
14.1. Deploying to the Cloud	562
14.1.1. Cloud Foundry	562
Binding to Services	564
14.1.2. Kubernetes	565
Kubernetes Container Lifecycle	565
14.1.3. Heroku	566

14.1.4. OpenShift	567
14.1.5. Amazon Web Services (AWS)	568
AWS Elastic Beanstalk	568
Summary	569
14.1.6. CloudCaptain and Amazon Web Services	569
14.1.7. Azure	570
14.1.8. Google Cloud	570
14.2. Installing Spring Boot Applications	571
14.2.1. Installation as a systemd Service	571
14.2.2. Installation as an init.d Service (System V)	572
Securing an init.d Service	574
Customizing the Startup Script	575
14.2.3. Microsoft Windows Services	578
14.3. Efficient deployments	578
14.3.1. Unpacking the Executable JAR	578
14.3.2. Using Ahead-of-time Processing With the JVM	578
14.3.3. Checkpoint and Restore With the JVM	579
14.4. What to Read Next	580
15. GraalVM Native Image Support	581
15.1. Introducing GraalVM Native Images	581
15.1.1. Key Differences with JVM Deployments	581
15.1.2. Understanding Spring Ahead-of-Time Processing	582
Source Code Generation	582
Hint File Generation	585
Proxy Class Generation	585
15.2. Developing Your First GraalVM Native Application	586
15.2.1. Sample Application	586
15.2.2. Building a Native Image Using Buildpacks	587
System Requirements	587
Using Maven	587
Using Gradle	588
Running the example	588
15.2.3. Building a Native Image using Native Build Tools	589
Prerequisites	589
Using Maven	589
Using Gradle	590
Running the Example	590
15.3. Testing GraalVM Native Images	591
15.3.1. Testing Ahead-of-time Processing With the JVM	591
15.3.2. Testing With Native Build Tools	592
Using Maven	592

Using Gradle	593
15.4. Advanced Native Images Topics	593
15.4.1. Nested Configuration Properties	593
15.4.2. Converting a Spring Boot Executable Jar	595
Using Buildpacks	595
Using GraalVM native-image	596
15.4.3. Using the Tracing Agent	597
Launch the Application Directly	597
15.4.4. Custom Hints	597
Testing custom hints	598
15.4.5. Known Limitations	599
15.5. What to Read Next	599
16. Spring Boot CLI	600
16.1. Installing the CLI	600
16.2. Using the CLI	600
16.2.1. Initialize a New Project	602
16.2.2. Using the Embedded Shell	603
17. Build Tool Plugins	604
17.1. Spring Boot Maven Plugin	604
17.2. Spring Boot Gradle Plugin	604
17.3. Spring Boot AntLib Module	604
17.3.1. Spring Boot Ant Tasks	605
Using the “exejar” Task	605
Examples	605
17.3.2. Using the “findmainclass” Task	606
Examples	606
17.4. Supporting Other Build Systems	606
17.4.1. Repackaging Archives	607
17.4.2. Nested Libraries	607
17.4.3. Finding a Main Class	607
17.4.4. Example Repackage Implementation	607
17.5. What to Read Next	609
18. “How-to” Guides	610
18.1. Spring Boot Application	610
18.1.1. Create Your Own FailureAnalyzer	610
18.1.2. Troubleshoot Auto-configuration	610
18.1.3. Customize the Environment or ApplicationContext Before It Starts	611
18.1.4. Build an ApplicationContext Hierarchy (Adding a Parent or Root Context)	614
18.1.5. Create a Non-web Application	614
18.2. Properties and Configuration	614
18.2.1. Automatically Expand Properties at Build Time	614

Automatic Property Expansion Using Maven	614
Automatic Property Expansion Using Gradle	615
18.2.2. Externalize the Configuration of SpringApplication	616
18.2.3. Change the Location of External Properties of an Application	619
18.2.4. Use ‘Short’ Command Line Arguments	619
18.2.5. Use YAML for External Properties	620
18.2.6. Set the Active Spring Profiles	620
18.2.7. Set the Default Profile Name	621
18.2.8. Change Configuration Depending on the Environment	621
18.2.9. Discover Built-in Options for External Properties	622
18.3. Embedded Web Servers	623
18.3.1. Use Another Web Server	623
18.3.2. Disabling the Web Server	624
18.3.3. Change the HTTP Port	624
18.3.4. Use a Random Unassigned HTTP Port	624
18.3.5. Discover the HTTP Port at Runtime	625
18.3.6. Enable HTTP Response Compression	626
18.3.7. Configure SSL	626
Using PEM-encoded files	627
18.3.8. Configure HTTP/2	628
HTTP/2 With Tomcat	628
HTTP/2 With Jetty	628
HTTP/2 With Reactor Netty	628
HTTP/2 With Undertow	629
18.3.9. Configure the Web Server	629
18.3.10. Add a Servlet, Filter, or Listener to an Application	630
Add a Servlet, Filter, or Listener by Using a Spring Bean	631
Add Servlets, Filters, and Listeners by Using Classpath Scanning	632
18.3.11. Configure Access Logging	632
18.3.12. Running Behind a Front-end Proxy Server	634
Customize Tomcat’s Proxy Configuration	634
18.3.13. Enable Multiple Connectors with Tomcat	635
18.3.14. Enable Tomcat’s MBean Registry	637
18.3.15. Enable Multiple Listeners with Undertow	638
18.3.16. Create WebSocket Endpoints Using @ServerEndpoint	639
18.4. Spring MVC	640
18.4.1. Write a JSON REST Service	640
18.4.2. Write an XML REST Service	641
18.4.3. Customize the Jackson ObjectMapper	642
18.4.4. Customize the @ResponseBody Rendering	644
18.4.5. Handling Multipart File Uploads	644

18.4.6. Switch Off the Spring MVC DispatcherServlet.....	645
18.4.7. Switch off the Default MVC Configuration	645
18.4.8. Customize ViewResolvers.....	645
18.5. Jersey	647
18.5.1. Secure Jersey endpoints with Spring Security.....	647
18.5.2. Use Jersey Alongside Another Web Framework.....	647
18.6. HTTP Clients	648
18.6.1. Configure RestTemplate to Use a Proxy	648
18.6.2. Configure the TcpClient used by a Reactor Netty-based WebClient	648
18.7. Logging	650
18.7.1. Configure Logback for Logging.....	651
Configure Logback for File-only Output	652
18.7.2. Configure Log4j for Logging	653
Use YAML or JSON to Configure Log4j 2	654
Use Composite Configuration to Configure Log4j 2	655
18.8. Data Access	655
18.8.1. Configure a Custom DataSource	655
18.8.2. Configure Two DataSources.....	662
18.8.3. Use Spring Data Repositories.....	668
18.8.4. Separate @Entity Definitions from Spring Configuration	668
18.8.5. Configure JPA Properties	669
18.8.6. Configure Hibernate Naming Strategy	670
18.8.7. Configure Hibernate Second-Level Caching	672
18.8.8. Use Dependency Injection in Hibernate Components	673
18.8.9. Use a Custom EntityManagerFactory.....	674
18.8.10. Using Multiple EntityManagerFactories.....	674
18.8.11. Use a Traditional persistence.xml File	678
18.8.12. Use Spring Data JPA and Mongo Repositories.....	678
18.8.13. Customize Spring Data's Web Support	679
18.8.14. Expose Spring Data Repositories as REST Endpoint	679
18.8.15. Configure a Component that is Used by JPA	679
18.8.16. Configure jOOQ with Two DataSources	680
18.9. Database Initialization	680
18.9.1. Initialize a Database Using Hibernate	681
18.9.2. Initialize a Database Using Basic SQL Scripts	681
18.9.3. Initialize a Spring Batch Database	682
18.9.4. Use a Higher-level Database Migration Tool	682
Execute Flyway Database Migrations on Startup	682
Execute Liquibase Database Migrations on Startup	684
Use Flyway for test-only migrations	685
Use Liquibase for test-only migrations	685

18.9.5. Depend Upon an Initialized Database	686
Detect a Database Initializer	686
Detect a Bean That Depends On Database Initialization	686
18.10. NoSQL	687
18.10.1. Use Jedis Instead of Lettuce	687
18.11. Messaging	687
18.11.1. Disable Transacted JMS Session	688
18.12. Batch Applications	689
18.12.1. Specifying a Batch Data Source	689
18.12.2. Running Spring Batch Jobs on Startup	689
18.12.3. Running From the Command Line	690
18.12.4. Restarting a stopped or failed Job	690
18.12.5. Storing the Job Repository	690
18.13. Actuator	690
18.13.1. Change the HTTP Port or Address of the Actuator Endpoints	690
18.13.2. Customize the ‘whitelabel’ Error Page	691
18.13.3. Customizing Sanitization	691
18.13.4. Map Health Indicators to Micrometer Metrics	691
18.14. Security	693
18.14.1. Switch off the Spring Boot Security Configuration	693
18.14.2. Change the UserDetailsService and Add User Accounts	693
18.14.3. Enable HTTPS When Running behind a Proxy Server	694
18.15. Hot Swapping	695
18.15.1. Reload Static Content	695
18.15.2. Reload Templates without Restarting the Container	696
Thymeleaf Templates	696
FreeMarker Templates	696
Groovy Templates	696
18.15.3. Fast Application Restarts	696
18.15.4. Reload Java Classes without Restarting the Container	696
18.16. Testing	697
18.16.1. Testing With Spring Security	697
18.16.2. Structure <code>@Configuration</code> classes for inclusion in slice tests	698
18.17. Build	700
18.17.1. Generate Build Information	701
18.17.2. Generate Git Information	701
18.17.3. Customize Dependency Versions	702
18.17.4. Create an Executable JAR with Maven	702
18.17.5. Use a Spring Boot Application as a Dependency	703
18.17.6. Extract Specific Libraries When an Executable Jar Runs	704
18.17.7. Create a Non-executable JAR with Exclusions	705

18.17.8. Remote Debug a Spring Boot Application Started with Maven	705
18.17.9. Build an Executable Archive From Ant without Using spring-boot-antlib	705
18.18. Ahead-of-time processing	706
18.18.1. Conditions	707
18.19. Traditional Deployment	708
18.19.1. Create a Deployable War File	708
18.19.2. Convert an Existing Application to Spring Boot	710
18.19.3. Deploying a WAR to WebLogic	715
18.20. Docker Compose	716
18.20.1. Customizing the JDBC URL	716
18.20.2. Sharing services between multiple applications	716
Appendices	718
Appendix A: Common Application Properties	718
.A.1. Core Properties	718
.A.2. Cache Properties	727
.A.3. Mail Properties	728
.A.4. JSON Properties	729
.A.5. Data Properties	731
.A.6. Transaction Properties	755
.A.7. Data Migration Properties	755
.A.8. Integration Properties	762
.A.9. Web Properties	790
.A.10. Templating Properties	800
.A.11. Server Properties	807
.A.12. Security Properties	820
.A.13. RSocket Properties	822
.A.14. Actuator Properties	824
.A.15. Devtools Properties	856
.A.16. Docker Compose Properties	857
.A.17. Testcontainers Properties	858
.A.18. Testing Properties	858
Appendix B: Configuration Metadata	858
.B.1. Metadata Format	858
Group Attributes	860
Property Attributes	861
Hint Attributes	863
Repeated Metadata Items	864
.B.2. Providing Manual Hints	865
Value Hint	865
Value Providers	866
.B.3. Generating Your Own Metadata by Using the Annotation Processor	872

Configuring the Annotation Processor	872
Automatic Metadata Generation.....	873
Adding Additional Metadata	879
Appendix C: Auto-configuration Classes	879
.C.1. spring-boot-autoconfigure	879
.C.2. spring-boot-actuator-autoconfigure.....	884
Appendix D: Test Auto-configuration Annotations	888
.D.1. Test Slices	888
Appendix E: The Executable Jar Format	898
.E.1. Nested JARs	898
The Executable Jar File Structure	898
The Executable War File Structure.....	899
Index Files	900
Classpath Index	900
Layer Index	900
.E.2. Spring Boot’s “NestedJarFile” Class	901
Compatibility With the Standard Java “JarFile”	901
.E.3. Launching Executable Jars.....	902
Launcher Manifest	902
.E.4. PropertiesLauncher Features	902
.E.5. Executable Jar Restrictions.....	904
.E.6. Alternative Single Jar Solutions	904
Appendix F: Dependency Versions	904
.F.1. Managed Dependency Coordinates	905
.F.2. Version Properties	946

This document is also available as [multiple HTML pages](#) and as [a single HTML page](#).

Chapter 1. Legal

Copyright © 2012-2024

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

Chapter 2. Getting Help

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

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

NOTE

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

Chapter 3. Documentation Overview

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

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

3.1. First Steps

If you are getting started with Spring Boot or 'Spring' in general, start with [the following topics](#):

- **From scratch:** [Overview](#) | [Requirements](#) | [Installation](#)
- **Tutorial:** [Part 1](#) | [Part 2](#)
- **Running your example:** [Part 1](#) | [Part 2](#)

3.2. Upgrading From an Earlier Version

You should always ensure that you are running a [supported version](#) of Spring Boot.

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

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

3.3. Developing With Spring Boot

Ready to actually start using Spring Boot? [We have you covered](#):

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

3.4. Learning About Spring Boot Features

Need more details about Spring Boot's core features? [The following content is for you](#):

- **Spring Application:** [SpringApplication](#)
- **External Configuration:** [External Configuration](#)
- **Profiles:** [Profiles](#)

- **Logging:** [Logging](#)

3.5. Web

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

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

3.6. Data

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

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

3.7. Messaging

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

- **JMS:** [Auto-configuration for ActiveMQ](#) and [Artemis](#), [Sending and Receiving messages through JMS](#)
- **AMQP:** [Auto-configuration for RabbitMQ](#)
- **Kafka:** [Auto-configuration for Spring Kafka](#)
- **Pulsar:** [Auto-configuration for Spring for Apache Pulsar](#)
- **RSocket:** [Auto-configuration for Spring Framework's RSocket Support](#)
- **Spring Integration:** [Auto-configuration for Spring Integration](#)

3.8. IO

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

- **Caching:** [Caching support with EhCache](#), [Hazelcast](#), [Infinispan](#), and more
- **Quartz:** [Quartz Scheduling](#)
- **Mail:** [Sending Email](#)
- **Validation:** [JSR-303 Validation](#)
- **REST Clients:** [Calling REST Services with RestTemplate](#) and [WebClient](#)

- **Webservices:** [Auto-configuration for Spring Web Services](#)
- **JTA:** [Distributed Transactions with JTA](#)

3.9. Container Images

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

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

3.10. Moving to Production

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

- **Management endpoints:** [Overview](#)
- **Connection options:** [HTTP | JMX](#)
- **Monitoring:** [Metrics | Auditing | HTTP Exchanges | Process](#)

3.11. GraalVM Native Images

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

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

3.12. Advanced Topics

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

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

Chapter 4. Getting Started

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

4.1. Introducing Spring Boot

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

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

Our primary goals are:

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

4.2. System Requirements

Spring Boot 3.2.7 requires [Java 17](#) and is compatible up to and including Java 22. [Spring Framework 6.1.10](#) or above is also required.

Explicit build support is provided for the following build tools:

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

4.2.1. Servlet Containers

Spring Boot supports the following embedded servlet containers:

Name	Servlet Version
Tomcat 10.1	6.0

Name	Servlet Version
Jetty 12.0	6.0
Undertow 2.3	6.0

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

4.2.2. GraalVM Native Images

Spring Boot applications can be [converted into a Native Image](#) using GraalVM 22.3 or above.

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

The following versions are supported:

Name	Version
GraalVM Community	22.3
Native Build Tools	0.9.28

4.3. Installing Spring Boot

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

```
$ java -version
```

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

4.3.1. Installation Instructions for the Java Developer

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

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

Maven Installation

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

TIP

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

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

More details on getting started with Spring Boot and Maven can be found in the [Getting Started section](#) of the Maven plugin’s reference guide.

Gradle Installation

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

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

Gradle Wrapper

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

More details on getting started with Spring Boot and Gradle can be found in the [Getting Started section](#) of the Gradle plugin’s reference guide.

4.3.2. Installing the Spring Boot CLI

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

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

Manual Installation

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

- [spring-boot-cli-3.2.7-bin.zip](#)
- [spring-boot-cli-3.2.7-bin.tar.gz](#)

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

correctly).

Installation with SDKMAN!

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

```
$ sdk install springboot  
$ spring --version  
Spring CLI v3.2.7
```

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

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-  
cli-3.2.7-bin/spring-3.2.7/  
$ sdk default springboot dev  
$ spring --version  
Spring CLI v3.2.7
```

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

You can see it by running the following command:

```
$ sdk ls springboot  
  
=====  
Available Springboot Versions  
=====  
> + dev  
* 3.2.7  
  
=====  
+ - local version  
* - installed  
> - currently in use  
=====
```

OSX Homebrew Installation

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

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

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

NOTE

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

MacPorts Installation

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

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

Command-line Completion

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

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

NOTE

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

Windows Scoop Installation

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

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

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

NOTE

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

4.4. Developing Your First Spring Boot Application

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

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

TIP You can shortcut the steps below by going to [start.spring.io](#) and choosing the “Web” starter from the dependencies searcher. Doing so generates a new project structure so that you can [start coding right away](#). Check the [start.spring.io user guide](#) for more details.

4.4.1. Prerequisites

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

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

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

Maven

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

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

Gradle

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

```
$ gradle --version

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

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

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

4.4.2. Setting up the project with Maven

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

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

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

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.7</version>
  </parent>

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

</project>
```

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

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

4.4.3. Setting up the project with Gradle

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

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.2.7'
}

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

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

repositories {
    mavenCentral()
}

dependencies {
```

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

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

4.4.4. Adding Classpath Dependencies

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

Maven

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

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

```
$ mvn dependency:tree
[INFO] com.example:myproject:jar:0.0.1-SNAPSHOT
```

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

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

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

Gradle

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

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

```
$ gradle dependencies
> Task :dependencies
-----
Root project 'myproject'
```

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

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

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

4.4.5. Writing the Code

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

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

Java

```
package com.example;

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

@RestController
@SpringBootApplication
public class MyApplication {

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

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

}
```

Kotlin

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

@RestController
@SpringBootApplication
class MyApplication {

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

}

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

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

The `@RestController` and `@RequestMapping` Annotations

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

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

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

The `@SpringBootApplication` Annotation

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

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

Starters and Auto-configuration

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

The “main” Method

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

4.4.6. Running the Example

Maven

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

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

Hello World!

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

Gradle

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

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

Hello World!

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

4.4.7. Creating an Executable Jar

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

Executable jars and Java

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

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

Spring Boot takes a [different approach](#) and lets you actually nest jars directly.

Maven

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

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

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

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

```
$ mvn package

[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building myproject 0.0.1-SNAPSHOT
[INFO] -----
[INFO] .... .
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ myproject ---
[INFO] Building jar: /Users/developer/example/spring-boot-example/target/myproject-
0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:3.2.7:repackage (default) @ myproject ---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

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

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

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

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

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

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

Gradle

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

```
$ gradle bootJar  
  
BUILD SUCCESSFUL in 639ms  
3 actionable tasks: 3 executed
```

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

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

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

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

```
\\ / _ _ _ ( ) _ _ _ \ \ \
( ( ) \ _ _ | ' _ | ' _ | ' _ \ \ ' | \ \ \
\ \ \ _ _ ) | ( _ ) | | | | | ( _ | ) ) ) )
' | _ _ | . _ _ | _ _ | _ \ _ , | / / / /
=====|_|=====|_/_=/_/_/_/
:: Spring Boot :: (v3.2.7)
..... .
..... (log output here)
..... .
..... Started MyApplication in 0.999 seconds (process running for 1.253)
```

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

4.5. What to Read Next

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

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

Chapter 5. Upgrading Spring Boot

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

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

5.1. Upgrading From 1.x

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

5.2. Upgrading to a New Feature Release

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

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

WARNING

Properties that are added late to the environment, such as when using [@PropertySource](#), will not be taken into account.

NOTE

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

5.3. Upgrading the Spring Boot CLI

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

5.4. What to Read Next

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

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

Chapter 6. Developing with Spring Boot

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

If you are starting out with Spring Boot, you should probably read the [Getting Started](#) guide before diving into this section.

6.1. Build Systems

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

6.1.1. Dependency Management

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

NOTE

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

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

WARNING

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

6.1.2. Maven

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

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

6.1.3. Gradle

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

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

6.1.4. Ant

It is possible to build a Spring Boot project using Apache Ant+Ivy. The [spring-boot-antlib](#) “AntLib” module is also available to help Ant create executable jars.

To declare dependencies, a typical [ivy.xml](#) file looks something like the following example:

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

A typical [build.xml](#) looks like the following example:

```

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

    <property name="spring-boot.version" value="3.2.7" />

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

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

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

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

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

TIP If you do not want to use the `spring-boot-antlib` module, see the [Build an Executable Archive From Ant without Using spring-boot-antlib](#) “How-to”.

6.1.5. Starters

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

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

What is in a name

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

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

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

Table 1. Spring Boot application starters

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

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

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

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

Table 2. Spring Boot production starters

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

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

Table 3. Spring Boot technical starters

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

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



For a list of additional community contributed starters, see the [README file](#) in the `spring-boot-starters` module on GitHub.

6.2. Structuring Your Code

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



If you wish to enforce a structure based on domains, take a look at [Spring Modular](#).

6.2.1. Using the “default” Package

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



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

6.2.2. Locating the Main Application Class

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

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

The following listing shows a typical layout:

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

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

Java

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

@SpringBootApplication
public class MyApplication {

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

}
```

```

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

@SpringBootApplication
class MyApplication

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

```

6.3. Configuration Classes

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

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

6.3.1. Importing Additional Configuration Classes

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

6.3.2. Importing XML Configuration

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

6.4. Auto-configuration

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

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

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

6.4.1. Gradually Replacing Auto-configuration

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

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

6.4.2. Disabling Specific Auto-configuration Classes

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

Java

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

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

}
```

Kotlin

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

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

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

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

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

6.4.3. Auto-configuration Packages

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

6.5. Spring Beans and Dependency Injection

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

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

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

Java

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

@Service
public class MyAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...
}
```

Kotlin

```
import org.springframework.stereotype.Service

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

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

Java

```
import java.io.PrintStream;

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

@Service
public class MyAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    private final PrintStream out;

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

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

    // ...
}
```

```

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

@Service
class MyAccountService : AccountService {

    private val riskAssessor: RiskAssessor

    private val out: PrintStream

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

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

    // ...
}

```

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

6.6. Using the `@SpringBootApplication` Annotation

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

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

Java

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

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

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

}
```

Kotlin

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

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

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

NOTE

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

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

Java

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

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

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

}
```

NOTE

Kotlin

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

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

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

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

6.7. Running Your Application

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

NOTE

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

6.7.1. Running From an IDE

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

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

TIP

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

6.7.2. Running as a Packaged Application

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

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

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

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

6.7.3. Using the Maven Plugin

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

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

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

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

6.7.4. Using the Gradle Plugin

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

```
$ gradle bootRun
```

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

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

6.7.5. Hot Swapping

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

The `spring-boot-devtools` module also includes support for quick application restarts. See the [Hot swapping “How-to”](#) for details.

6.8. Developer Tools

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

Maven

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

Gradle

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

CAUTION

Devtools might cause classloading issues, in particular in multi-module projects.

[Diagnosing Classloading Issues](#) explains how to diagnose and solve them.

NOTE

Developer tools are automatically disabled when running a fully packaged application. If your application is launched from `java -jar` or if it is started from a special classloader, then it is considered a “production application”. You can control this behavior by using the `spring.devtools.restart.enabled` system property. To enable devtools, irrespective of the classloader used to launch your application, set the `-Dspring.devtools.restart.enabled=true` system property. This must not be done in a production environment where running devtools is a security risk. To disable devtools, exclude the dependency or set the `-Dspring.devtools.restart.enabled=false` system property.

TIP

Flagging the dependency as optional in Maven or using the `developmentOnly` configuration in Gradle (as shown above) prevents devtools from being transitively applied to other modules that use your project.

TIP

Repackaged archives do not contain devtools by default. If you want to use a [certain remote devtools feature](#), you need to include it. When using the Maven plugin, set the `excludeDevtools` property to `false`. When using the Gradle plugin, [configure the task's classpath to include the developmentOnly configuration](#).

6.8.1. Diagnosing Classloading Issues

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

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

6.8.2. Property Defaults

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

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

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

The following table lists all the properties that are applied:

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

NOTE

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

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

6.8.3. Automatic Restart

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

Triggering a restart

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

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

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

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

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

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

CAUTION Automatic restart is not supported when using AspectJ weaving.

Restart vs Reload

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

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

Logging Changes in Condition Evaluation

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

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

Properties

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

Yaml

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

Excluding Resources

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

Properties

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

Yaml

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

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

Watching Additional Paths

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

Disabling Restart

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

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

Java

```
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class MyApplication {  
  
    public static void main(String[] args) {  
        System.setProperty("spring.devtools.restart.enabled", "false");  
        SpringApplication.run(MyApplication.class, args);  
    }  
}
```

Kotlin

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

@SpringBootApplication
object MyApplication {

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

}
```

Using a Trigger File

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

NOTE

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

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

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

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

Then your `trigger-file` property would be:

Properties

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

Yaml

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

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

TIP

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

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

Customizing the Restart Classloader

As described earlier in the [Restart vs Reload](#) section, restart functionality is implemented by using two classloaders. If this causes issues, you can diagnose the problem by using the `spring.devtools.restart.enabled` system property, and if the app works with restart switched off, you might need to customize what gets loaded by which classloader.

By default, any open project in your IDE is loaded with the “restart” classloader, and any regular `.jar` file is loaded with the “base” classloader. The same is true if you use `mvn spring-boot:run` or `gradle bootRun`: the project containing your `@SpringBootApplication` is loaded with the “restart” classloader, and everything else with the “base” classloader. The classpath is printed on the console when you start the app, which can help to identify any problematic entries. Classes used reflectively, especially annotations, can be loaded into the parent (fixed) classloader on startup before the application classes which uses them, and this might lead to them not being detected by Spring in the application.

You can instruct Spring Boot to load parts of your project with a different classloader by creating a `META-INF/spring-devtools.properties` file. The `spring-devtools.properties` file can contain properties prefixed with `restart.exclude` and `restart.include`. The `include` elements are items that should be pulled up into the “restart” classloader, and the `exclude` elements are items that should be pushed down into the “base” classloader. The value of the property is a regex pattern that is applied to the classpath passed to the JVM on startup. Here is an example where some local class files are excluded and some extra libraries are included in the restart class loader:

Properties

```
restart.exclude.companycommonlibs=/mycorp-common-[\\w\\d-\\.]/(build|bin|out|target)/  
restart.include.projectcommon=/mycorp-myproj-[\\w\\d-\\.]+\\.jar
```

Yaml

```
restart:  
  exclude:  
    companycommonlibs: "/mycorp-common-[\\w\\d-\\.]/(build|bin|out|target)/*"  
  include:  
    projectcommon: "/mycorp-myproj-[\\w\\d-\\.]+\\.jar"
```

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

TIP All `META-INF/spring-devtools.properties` from the classpath are loaded. You can package files inside your project, or in the libraries that the project consumes. System properties can not be used, only the properties file.

Known Limitations

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

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

6.8.4. LiveReload

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

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

NOTE You can only run one LiveReload server at a time. Before starting your application, ensure that no other LiveReload servers are running. If you start multiple applications from your IDE, only the first has LiveReload support.

WARNING To trigger LiveReload when a file changes, [Automatic Restart](#) must be enabled.

6.8.5. Global Settings

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

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

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

Properties

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

Yaml

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

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

NOTE

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

NOTE

Profiles are not supported in devtools properties/yaml files.

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

Configuring File System Watcher

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

Properties

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

Yaml

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

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

6.8.6. Remote Applications

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

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

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

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

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

NOTE Remote devtools is not supported for Spring WebFlux applications.

Running the Remote Client Application

The remote client application is designed to be run from within your IDE. You need to run `org.springframework.boot.devtools.RemoteSpringApplication` with the same classpath as the remote

project that you connect to. The application's single required argument is the remote URL to which it connects.

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

- Select `Run Configurations...` from the `Run` menu.
 - Create a new `Java Application` “launch configuration”.
 - Browse for the `my-app` project.
 - Use `org.springframework.boot.devtools.RemoteSpringApplication` as the main class.
 - Add `https://myapp.cfapps.io` to the `Program arguments` (or whatever your remote URL is).

A running remote client might resemble the following listing:

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

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

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

Remote Update

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

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

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

NOTE

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

6.9. Packaging Your Application for Production

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

For additional “production ready” features, such as health, auditing, and metric REST or JMX endpoints, consider adding [spring-boot-actuator](#). See [Production-ready Features](#) for details.

6.10. What to Read Next

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

Chapter 7. Core Features

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

7.1. SpringApplication

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

Java

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

@SpringBootApplication
public class MyApplication {

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

}
```

Kotlin

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

@SpringBootApplication
class MyApplication

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

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

```
.-'-----'  
\\ / _-'_--_(_)_--_--_\\ \\ \\  
( ( )\__| '_| '_| '_\\` | \\ \\  
\\ \\_ __)|_|_|_|_|_|(|_|_) ) ) )  
' |_____| .__|_|_|_|_|_\_, | / / / /  
=====|_|=====|_|=/_/_/_/  
:: Spring Boot :: (v3.2.7)
```

```
2024-06-20T08:04:34.953Z INFO 112644 --- [           main]  
o.s.b.d.f.logexample.MyApplication      : Starting MyApplication using Java 17.0.11  
with PID 112644 (/opt/apps/myapp.jar started by myuser in /opt/apps/)  
2024-06-20T08:04:35.034Z INFO 112644 --- [           main]  
o.s.b.d.f.logexample.MyApplication      : No active profile set, falling back to 1  
default profile: "default"  
2024-06-20T08:04:39.936Z INFO 112644 --- [           main]  
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)  
2024-06-20T08:04:40.030Z INFO 112644 --- [           main]  
o.apache.catalina.core.StandardService   : Starting service [Tomcat]  
2024-06-20T08:04:40.039Z INFO 112644 --- [           main]  
o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache  
Tomcat/10.1.25]  
2024-06-20T08:04:40.281Z INFO 112644 --- [           main]  
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded  
WebApplicationContext  
2024-06-20T08:04:40.308Z INFO 112644 --- [           main]  
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization  
completed in 5028 ms  
2024-06-20T08:04:42.197Z INFO 112644 --- [           main]  
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with  
context path ''  
2024-06-20T08:04:42.219Z INFO 112644 --- [           main]  
o.s.b.d.f.logexample.MyApplication      : Started MyApplication in 9.396 seconds  
(process running for 10.948)
```

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

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

7.1.1. Startup Failure

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

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

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

Description:

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

Action:

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

NOTE

Spring Boot provides numerous `FailureAnalyzer` implementations, and you can [add your own](#).

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

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

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

7.1.2. Lazy Initialization

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

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

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

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

Properties

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

Yaml

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

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

7.1.3. Customizing the Banner

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

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

Table 4. Banner variables

Variable	Description
<code> \${application.version}</code>	The version number of your application, as declared in <code>MANIFEST.MF</code> . For example, <code>Implementation-Version: 1.0</code> is printed as <code>1.0</code> .
<code> \${application.formatted-version}</code>	The version number of your application, as declared in <code>MANIFEST.MF</code> and formatted for display (surrounded with brackets and prefixed with <code>v</code>). For example <code>(v1.0)</code> .
<code> \${spring-boot.version}</code>	The Spring Boot version that you are using. For example <code>3.2.7</code> .
<code> \${spring-boot.formatted-version}</code>	The Spring Boot version that you are using, formatted for display (surrounded with brackets and prefixed with <code>v</code>). For example <code>(v3.2.7)</code> .
<code> \${Ansi.NAME}</code> (or <code> \${AnsiColor.NAME}</code> , <code> \${Ansibackground.NAME}</code> , <code> \${Ansistyle.NAME}</code>)	Where <code>NAME</code> is the name of an ANSI escape code. See AnsiPropertySource for details.
<code> \${application.title}</code>	The title of your application, as declared in <code>MANIFEST.MF</code> . For example <code>Implementation-Title: MyApp</code> is printed as <code>MyApp</code> .

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

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

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

The `application.title`, `application.version`, and `application.formatted-version` properties are only available if you are using `java -jar` or `java -cp` with Spring Boot launchers. The values will not be resolved if you are running an unpacked jar and starting it with `java -cp <classpath> <mainclass>` or running your application as a native image.

NOTE

To use the `application.properties`, launch your application as a packed jar using `java -jar` or as an unpacked jar using `java org.springframework.boot.loader.launch.JarLauncher`. This will initialize the `application.banner` properties before building the classpath and launching your app.

7.1.4. Customizing SpringApplication

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

Java

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

@SpringBootApplication
public class MyApplication {

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

Kotlin

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

@SpringBootApplication
class MyApplication

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

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

It is also possible to configure the `SpringApplication` by using an `application.properties` file. See [Externalized Configuration](#) for details.

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

7.1.5. Fluent Builder API

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

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

Java

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

Kotlin

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

NOTE

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

7.1.6. Application Availability

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

In addition, you can also obtain availability states by injecting the [ApplicationAvailability](#) interface into your own beans.

Liveness State

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

NOTE

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

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

Readiness State

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

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

TIP

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

Managing the Application Availability State

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

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

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

Java

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

@Component
public class MyReadinessStateExporter {

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

Kotlin

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

@Component
class MyReadinessStateExporter {

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

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

Java

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

@Component
public class MyLocalCacheVerifier {

    private final ApplicationEventPublisher eventPublisher;

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

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

Kotlin

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

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

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

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

the dedicated section.

7.1.7. Application Events and Listeners

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

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

NOTE

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

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

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

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

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

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

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

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

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

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

7.1.8. Web Environment

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

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

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

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

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

7.1.9. Accessing Application Arguments

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

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

Java

```
import java.util.List;

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

@Component
public class MyBean {

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

}
```

Kotlin

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

@Component
class MyBean(args: ApplicationArguments) {

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

}
```

TIP

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

7.1.10. Using the `ApplicationRunner` or `CommandLineRunner`

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

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

NOTE

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

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

Java

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

@Component
public class MyCommandLineRunner implements CommandLineRunner {

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

}
```

Kotlin

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

@Component
class MyCommandLineRunner : CommandLineRunner {

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

}
```

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

7.1.11. Application Exit

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

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

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

Java

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

@SpringBootApplication
public class MyApplication {

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

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

}
```

Kotlin

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

import kotlin.system.exitProcess

@SpringBootApplication
class MyApplication {

    @Bean
    fun exitCodeGenerator() = ExitCodeGenerator { 42 }

}

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

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

method.

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

7.1.12. Admin Features

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

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

7.1.13. Application Startup tracking

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

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

Java

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

@SpringBootApplication
public class MyApplication {

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

```

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

@SpringBootApplication
class MyApplication

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

```

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

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

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

Spring Boot can also be configured to expose a [startup endpoint](#) that provides this information as a JSON document.

7.1.14. Virtual threads

If you're running on Java 21 or up, you can enable virtual threads by setting the property [spring.threads.virtual.enabled](#) to [true](#).

Before turning on this option for your application, you should consider [reading the official Java virtual threads documentation](#). In some cases, applications can experience lower throughput because of "Pinned Virtual Threads"; this page also explains how to detect such cases with JDK Flight Recorder or the [jcmd](#) CLI.

WARNING

One side effect of virtual threads is that they are daemon threads. A JVM will exit if all of its threads are daemon threads. This behavior can be a problem when you rely on `@Scheduled` beans, for example, to keep your application alive. If you use virtual threads, the scheduler thread is a virtual thread and therefore a daemon thread and won't keep the JVM alive. This not only affects scheduling and can be the case with other technologies too. To keep the JVM running in all cases, it is recommended to set the property `spring.main.keep-alive` to `true`. This ensures that the JVM is kept alive, even if all threads are virtual threads.

7.2. Externalized Configuration

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

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

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

1. Default properties (specified by setting `SpringApplication.setDefaultProperties`).
2. `@PropertySource` annotations on your `@Configuration` classes. Please note that such property sources are not added to the `Environment` until the application context is being refreshed. This is too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.
3. Config data (such as `application.properties` files).
4. A `RandomValuePropertySource` that has properties only in `random.*`.
5. OS environment variables.
6. Java System properties (`System.getProperties()`).
7. JNDI attributes from `java:comp/env`.
8. `ServletContext` init parameters.
9. `ServletConfig` init parameters.
10. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. `properties` attribute on your tests. Available on `@SpringBootTest` and the `test` annotations for testing a particular slice of your application.
13. `@DynamicPropertySource` annotations in your tests.
14. `@TestPropertySource` annotations on your tests.

15. Devtools global settings properties in the `$HOME/.config/spring-boot` directory when devtools is active.

Config data files are considered in the following order:

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

NOTE It is recommended to stick with one format for your entire application. If you have configuration files with both `.properties` and YAML format in the same location, `.properties` takes precedence.

NOTE If you use environment variables rather than system properties, most operating systems disallow period-separated key names, but you can use underscores instead (for example, `SPRING_CONFIG_NAME` instead of `spring.config.name`). See [Binding From Environment Variables](#) for details.

NOTE If your application runs in a servlet container or application server, then JNDI properties (in `java:comp/env`) or servlet context initialization parameters can be used instead of, or as well as, environment variables or system properties.

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

Java

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

@Component
public class MyBean {

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

    // ...

}
```

```

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

@Component
class MyBean {

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

    // ...

}

```

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

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

7.2.1. Accessing Command Line Properties

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

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

7.2.2. JSON Application Properties

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

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

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

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

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

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

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

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

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

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

NOTE

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

7.2.3. External Application Properties

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

1. From the classpath
 - a. The classpath root
 - b. The classpath `/config` package
2. From the current directory
 - a. The current directory
 - b. The `config/` subdirectory in the current directory
 - c. Immediate child directories of the `config/` subdirectory

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

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

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

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

The following example shows how to specify two distinct files:

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

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

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

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

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

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

If you have a complex location setup, and you use profile-specific configuration files, you may need to provide further hints so that Spring Boot knows how they should be grouped. A location group is a collection of locations that are all considered at the same level. For example, you might want to group all classpath locations, then all external locations. Items within a location group should be separated with `;`. See the example in the “Profile Specific Files” section for more details.

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

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

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

1. `optional:classpath:/;optional:classpath:/config/`
2. `optional:file:./;optional:file:./config/;optional:file:./config/*/`

3. `optional:classpath:custom-config/`

4. `optional:file:./custom-config/`

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

Optional Locations

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

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

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

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

Wildcard Locations

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

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

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

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

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

TIP

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

Profile Specific Files

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

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

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

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

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

NOTE

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

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

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

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

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

NOTE

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

Importing Additional Data

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

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

Properties

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

Yaml

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

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

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

Properties

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

Yaml

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

Properties

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

Yaml

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

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

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

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

Spring Boot includes pluggable API that allows various different location addresses to be supported. By default you can import Java Properties, YAML and “[configuration trees](#)”.

TIP Third-party jars can offer support for additional technologies (there is no requirement for files to be local). For example, you can imagine config data being from external stores such as Consul, Apache ZooKeeper or Netflix Archaius.

If you want to support your own locations, see the `ConfigDataLocationResolver` and `ConfigDataLoader` classes in the `org.springframework.boot.context.config` package.

Importing Extensionless Files

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

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

Properties

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

Yaml

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

Using Configuration Trees

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

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

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

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

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

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

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

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

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

Properties

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

Yaml

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

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

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

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

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

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

For example, given the following volume:

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

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

Properties

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

Yaml

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

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

NOTE

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

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

Properties

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

Yaml

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

Property Placeholders

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

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

Properties

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

Yaml

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

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

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

NOTE

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

TIP

You can also use this technique to create “short” variants of existing Spring Boot properties. See the [Use ‘Short’ Command Line Arguments](#) how-to for details.

Working With Multi-Document Files

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

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

For example, the following file has two logical documents:

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

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

```

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

```

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

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

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

Activation Properties

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

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

The following activation properties are available:

Table 5. activation properties

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

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

Properties

```

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

```

```

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

```

7.2.4. Encrypting Properties

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

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

7.2.5. Working With YAML

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

NOTE If you use “Starters”, SnakeYAML is automatically provided by [spring-boot-starter](#).

Mapping YAML to Properties

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

```

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

```

In order to access these properties from the [Environment](#), they would be flattened as follows:

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

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

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

The preceding example would be transformed into these properties:

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

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

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

Directly Loading YAML

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

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

7.2.6. Configuring Random Values

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

Properties

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

Yaml

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

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

7.2.7. Configuring System Environment Properties

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

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

7.2.8. Type-safe Configuration Properties

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

TIP See also the [differences between @Value and type-safe configuration properties](#).

JavaBean Properties Binding

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

Java

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

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

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

    private boolean enabled;

    private InetAddress remoteAddress;

    private final Security security = new Security();

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

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

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

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

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

    public static class Security {

        private String username;

        private String password;

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

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

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

```

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

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

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

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

}

}

```

Kotlin

```

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

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

    var isEnabled = false

    var remoteAddress: InetAddress? = null

    val security = Security()

    class Security {

        var username: String? = null

        var password: String? = null

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

    }

}

```

The preceding POJO defines the following properties:

- `my.service.enabled`, with a value of `false` by default.

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

NOTE

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

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

NOTE

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

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

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

Constructor Binding

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

Java

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

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

```

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

    private final boolean enabled;

    private final InetAddress remoteAddress;

    private final Security security;

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

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

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

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

    public static class Security {

        private final String username;

        private final String password;

        private final List<String> roles;

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

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

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

```

```

    }

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

}

}

```

Kotlin

```

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

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

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

```

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

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

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

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

Java

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

Kotlin

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

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

}
```

NOTE To use constructor binding the class must be enabled using `@EnableConfigurationProperties` or configuration property scanning. You cannot use constructor binding with beans that are created by the regular Spring mechanisms (for example `@Component` beans, beans created by using `@Bean` methods or beans loaded by using `@Import`)

NOTE To use constructor binding the class must be compiled with `-parameters`. This will happen automatically if you use Spring Boot's Gradle plugin or if you use Maven and `spring-boot-starter-parent`.

NOTE The use of `java.util.Optional` with `@ConfigurationProperties` is not recommended as it is primarily intended for use as a return type. As such, it is not well-suited to configuration property injection. For consistency with properties of other types, if you do declare an `Optional` property and it has no value, `null` rather than an empty `Optional` will be bound.

Enabling `@ConfigurationProperties`-annotated Types

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

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

Java

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

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

}
```

Kotlin

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

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

Java

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

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

}
```

Kotlin

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

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

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

Java

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

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

}
```

Kotlin

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

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

NOTE

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

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

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

Using `@ConfigurationProperties`-annotated Types

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

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

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

Java

```
import org.springframework.stereotype.Service;  
  
@Service  
public class MyService {  
  
    private final MyProperties properties;  
  
    public MyService(MyProperties properties) {  
        this.properties = properties;  
    }  
  
    public void openConnection() {  
        Server server = new Server(this.properties.getRemoteAddress());  
        server.start();  
        // ...  
    }  
  
    // ...  
}
```

```

import org.springframework.stereotype.Service

@Service
class MyService(val properties: MyProperties) {

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

    // ...
}

}

```

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

Third-party Configuration

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

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

Java

```

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

@Configuration(proxyBeanMethods = false)
public class ThirdPartyConfiguration {

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

```

Kotlin

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

@Configuration(proxyBeanMethods = false)
class ThirdPartyConfiguration {

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

}
```

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

Relaxed Binding

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

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

Java

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

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

    private String firstName;

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

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

}
```

```
import org.springframework.boot.context.properties.ConfigurationProperties
@ConfigurationProperties(prefix = "my.main-project.person")
class MyPersonProperties {

    var firstName: String? = null
}
```

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

Table 6. relaxed binding

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

NOTE

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

Table 7. relaxed binding rules per property source

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

TIP

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

Binding Maps

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

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

Properties

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

Yaml

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

NOTE

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

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

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

Binding From Environment Variables

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

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

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

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

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

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

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

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

Caching

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

Merging Complex Types

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

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

Java

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

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

@ConfigurationProperties("my")
public class MyProperties {

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

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

}
```

Kotlin

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

@ConfigurationProperties("my")
class MyProperties {

    val list: List<MyPojo> = ArrayList()

}
```

Consider the following configuration:

Properties

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

Yaml

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

If the `dev` profile is not active, `MyProperties.list` contains one `MyPojo` entry, as previously defined. If the `dev` profile is enabled, however, the `list` still contains only one entry (with a name of `my another name` and a description of `null`). This configuration does not add a second `MyPojo` instance to the list, and it does not merge the items.

When a `List` is specified in multiple profiles, the one with the highest priority (and only that one) is used. Consider the following example:

Properties

```
my.list[0].name=my name
my.list[0].description=my description
my.list[1].name=another name
my.list[1].description=another description
#---
spring.config.activate.on-profile=dev
my.list[0].name=my another name
```

Yaml

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

In the preceding example, if the `dev` profile is active, `MyProperties.list` contains *one* `MyPojo` entry (with a name of `my another name` and a description of `null`). For YAML, both comma-separated lists and YAML lists can be used for completely overriding the contents of the list.

For `Map` properties, you can bind with property values drawn from multiple sources. However, for the same property in multiple sources, the one with the highest priority is used. The following example exposes a `Map<String, MyPojo>` from `MyProperties`:

Java

```
import java.util.LinkedHashMap;
import java.util.Map;

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

@ConfigurationProperties("my")
public class MyProperties {

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

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

}
```

Kotlin

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

@ConfigurationProperties("my")
class MyProperties {

    val map: Map<String, MyPojo> = LinkedHashMap()

}
```

Consider the following configuration:

Properties

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

```

my:
  map:
    key1:
      name: "my name 1"
      description: "my description 1"
---
spring:
  config:
    activate:
      on-profile: "dev"
my:
  map:
    key1:
      name: "dev name 1"
    key2:
      name: "dev name 2"
      description: "dev description 2"

```

If the `dev` profile is not active, `MyProperties.map` contains one entry with key `key1` (with a name of `my name 1` and a description of `my description 1`). If the `dev` profile is enabled, however, `map` contains two entries with keys `key1` (with a name of `dev name 1` and a description of `my description 1`) and `key2` (with a name of `dev name 2` and a description of `dev description 2`).

NOTE

The preceding merging rules apply to properties from all property sources, and not just files.

Properties Conversion

Spring Boot attempts to coerce the external application properties to the right type when it binds to the `@ConfigurationProperties` beans. If you need custom type conversion, you can provide a `ConversionService` bean (with a bean named `conversionService`) or custom property editors (through a `CustomEditorConfigurer` bean) or custom `Converters` (with bean definitions annotated as `@ConfigurationPropertiesBinding`).

NOTE

As this bean is requested very early during the application lifecycle, make sure to limit the dependencies that your `ConversionService` is using. Typically, any dependency that you require may not be fully initialized at creation time. You may want to rename your custom `ConversionService` if it is not required for configuration keys coercion and only rely on custom converters qualified with `@ConfigurationPropertiesBinding`.

Converting Durations

Spring Boot has dedicated support for expressing durations. If you expose a `java.time.Duration` property, the following formats in application properties are available:

- A regular `long` representation (using milliseconds as the default unit unless a `@DurationUnit` has

been specified)

- The standard ISO-8601 format used by `java.time.Duration`
- A more readable format where the value and the unit are coupled (`10s` means 10 seconds)

Consider the following example:

Java

```
import java.time.Duration;
import java.time.temporal.ChronoUnit;

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

@ConfigurationProperties("my")
public class MyProperties {

    @DurationUnit(ChronoUnit.SECONDS)
    private Duration sessionTimeout = Duration.ofSeconds(30);

    private Duration readTimeout = Duration.ofMillis(1000);

    public Duration getSessionTimeout() {
        return this.sessionTimeout;
    }

    public void setSessionTimeout(Duration sessionTimeout) {
        this.sessionTimeout = sessionTimeout;
    }

    public Duration getReadTimeout() {
        return this.readTimeout;
    }

    public void setReadTimeout(Duration readTimeout) {
        this.readTimeout = readTimeout;
    }

}
```

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.convert.DurationUnit
import java.time.Duration
import java.time.temporal.ChronoUnit

@ConfigurationProperties("my")
class MyProperties {

    @DurationUnit(ChronoUnit.SECONDS)
    var sessionTimeout = Duration.ofSeconds(30)

    var readTimeout = Duration.ofMillis(1000)

}
```

To specify a session timeout of 30 seconds, `30`, `PT30S` and `30s` are all equivalent. A read timeout of 500ms can be specified in any of the following form: `500`, `PT0.5S` and `500ms`.

You can also use any of the supported units. These are:

- `ns` for nanoseconds
- `us` for microseconds
- `ms` for milliseconds
- `s` for seconds
- `m` for minutes
- `h` for hours
- `d` for days

The default unit is milliseconds and can be overridden using `@DurationUnit` as illustrated in the sample above.

If you prefer to use constructor binding, the same properties can be exposed, as shown in the following example:

Java

```
import java.time.Duration;
import java.time.temporal.ChronoUnit;

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

@ConfigurationProperties("my")
public class MyProperties {

    private final Duration sessionTimeout;

    private final Duration readTimeout;

    public MyProperties(@DurationUnit(ChronoUnit.SECONDS) @DefaultValue("30s")
Duration sessionTimeout,
                        @DefaultValue("1000ms") Duration readTimeout) {
        this.sessionTimeout = sessionTimeout;
        this.readTimeout = readTimeout;
    }

    public Duration getSessionTimeout() {
        return this.sessionTimeout;
    }

    public Duration getReadTimeout() {
        return this.readTimeout;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.context.properties.bind.DefaultValue
import org.springframework.boot.convert.DurationUnit
import java.time.Duration
import java.time.temporal.ChronoUnit

@ConfigurationProperties("my")
class MyProperties(@param:DurationUnit(ChronoUnit.SECONDS) @param:DefaultValue("30s")
val sessionTimeout: Duration,
                    @param:DefaultValue("1000ms") val readTimeout: Duration)
```

TIP If you are upgrading a `Long` property, make sure to define the unit (using `@DurationUnit`) if it is not milliseconds. Doing so gives a transparent upgrade path while supporting a much richer format.

Converting Periods

In addition to durations, Spring Boot can also work with `java.time.Period` type. The following formats can be used in application properties:

- A regular `int` representation (using days as the default unit unless a `@PeriodUnit` has been specified)
- The standard ISO-8601 format [used by `java.time.Period`](#)
- A simpler format where the value and the unit pairs are coupled (`1y3d` means 1 year and 3 days)

The following units are supported with the simple format:

- `y` for years
- `m` for months
- `w` for weeks
- `d` for days

NOTE

The `java.time.Period` type never actually stores the number of weeks, it is a shortcut that means “7 days”.

Converting Data Sizes

Spring Framework has a `DataSize` value type that expresses a size in bytes. If you expose a `DataSize` property, the following formats in application properties are available:

- A regular `long` representation (using bytes as the default unit unless a `@DataSizeUnit` has been specified)
- A more readable format where the value and the unit are coupled (`10MB` means 10 megabytes)

Consider the following example:

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.convert.DataSizeUnit;
import org.springframework.util.unit.DataSize;
import org.springframework.util.unit.DataUnit;

@ConfigurationProperties("my")
public class MyProperties {

    @DataSizeUnit(DataUnit.MEGABYTES)
    private DataSize bufferSize = DataSize.ofMegabytes(2);

    private DataSize sizeThreshold = DataSize.ofBytes(512);

    public DataSize getBufferSize() {
        return this.bufferSize;
    }

    public void setBufferSize(DataSize bufferSize) {
        this.bufferSize = bufferSize;
    }

    public DataSize getSizeThreshold() {
        return this.sizeThreshold;
    }

    public void setSizeThreshold(DataSize sizeThreshold) {
        this.sizeThreshold = sizeThreshold;
    }

}
```

Kotlin

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.convert.DataSizeUnit
import org.springframework.util.unit.DataSize
import org.springframework.util.unit.DataUnit

@ConfigurationProperties("my")
class MyProperties {

    @DataSizeUnit(DataUnit.MEGABYTES)
    var bufferSize = DataSize.ofMegabytes(2)

    var sizeThreshold = DataSize.ofBytes(512)

}
```

To specify a buffer size of 10 megabytes, `10` and `10MB` are equivalent. A size threshold of 256 bytes can be specified as `256` or `256B`.

You can also use any of the supported units. These are:

- `B` for bytes
- `KB` for kilobytes
- `MB` for megabytes
- `GB` for gigabytes
- `TB` for terabytes

The default unit is bytes and can be overridden using `@DataSizeUnit` as illustrated in the sample above.

If you prefer to use constructor binding, the same properties can be exposed, as shown in the following example:

Java

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.bind.DefaultValue;
import org.springframework.boot.convert.DataSizeUnit;
import org.springframework.util.unit.DataSize;
import org.springframework.util.unit.DataUnit;

@ConfigurationProperties("my")
public class MyProperties {

    private final DataSize bufferSize;

    private final DataSize sizeThreshold;

    public MyProperties(@DataSizeUnit(DataUnit.MEGABYTES) @DefaultValue("2MB")
DataSize bufferSize,
                        @DefaultValue("512B") DataSize sizeThreshold) {
        this.bufferSize = bufferSize;
        this.sizeThreshold = sizeThreshold;
    }

    public DataSize getBufferSize() {
        return this.bufferSize;
    }

    public DataSize getSizeThreshold() {
        return this.sizeThreshold;
    }

}
```

```
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.boot.context.properties.bind.DefaultValue
import org.springframework.boot.convert.DataSizeUnit
import org.springframework.util.unit.DataSize
import org.springframework.util.unit.DataUnit

@ConfigurationProperties("my")
class MyProperties(@param:DataSizeUnit(DataUnit.MEGABYTES) @param:DefaultValue("2MB")
val bufferSize: DataSize,
    @param:DefaultValue("512B") val sizeThreshold: DataSize)
```

TIP If you are upgrading a `Long` property, make sure to define the unit (using `@DataSizeUnit`) if it is not bytes. Doing so gives a transparent upgrade path while supporting a much richer format.

@ConfigurationProperties Validation

Spring Boot attempts to validate `@ConfigurationProperties` classes whenever they are annotated with Spring's `@Validated` annotation. You can use JSR-303 `jakarta.validation` constraint annotations directly on your configuration class. To do so, ensure that a compliant JSR-303 implementation is on your classpath and then add constraint annotations to your fields, as shown in the following example:

Java

```
import java.net.InetAddress;  
  
import jakarta.validation.constraints.NotNull;  
  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.validation.annotation.Validated;  
  
{@ConfigurationProperties("my.service")  
@Validated  
public class MyProperties {  
  
    @NotNull  
    private InetAddress remoteAddress;  
  
    public InetAddress getRemoteAddress() {  
        return this.remoteAddress;  
    }  
  
    public void setRemoteAddress(InetAddress remoteAddress) {  
        this.remoteAddress = remoteAddress;  
    }  
}
```

Kotlin

```
import jakarta.validation.constraints.NotNull  
import org.springframework.boot.context.properties.ConfigurationProperties  
import org.springframework.validation.annotation.Validated  
import java.net.InetAddress  
  
{@ConfigurationProperties("my.service")  
@Validated  
class MyProperties {  
  
    var remoteAddress: @NotNull InetAddress? = null  
}
```

TIP You can also trigger validation by annotating the `@Bean` method that creates the configuration properties with `@Validated`.

To ensure that validation is always triggered for nested properties, even when no properties are found, the associated field must be annotated with `@Valid`. The following example builds on the preceding `MyProperties` example:

Java

```
import java.net.InetAddress;

import jakarta.validation.Valid;
import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.NotNull;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.validation.annotation.Validated;

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

    @NotNull
    private InetAddress remoteAddress;

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

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

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

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

    public static class Security {

        @NotEmpty
        private String username;

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

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

    }
}
```

```

import jakarta.validation.Valid
import jakarta.validation.constraints.NotEmpty
import jakarta.validation.constraints.NotNull
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.validation.annotation.Validated
import java.net.InetAddress

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

    var remoteAddress: @NotNull InetAddress? = null

    @Valid
    val security = Security()

    class Security {

        @NotEmpty
        var username: String? = null

    }
}

```

You can also add a custom Spring `Validator` by creating a bean definition called `configurationPropertiesValidator`. The `@Bean` method should be declared `static`. The configuration properties validator is created very early in the application's lifecycle, and declaring the `@Bean` method as static lets the bean be created without having to instantiate the `@Configuration` class. Doing so avoids any problems that may be caused by early instantiation.

TIP The `spring-boot-actuator` module includes an endpoint that exposes all `@ConfigurationProperties` beans. Point your web browser to `/actuator/configprops` or use the equivalent JMX endpoint. See the "Production ready features" section for details.

@ConfigurationProperties vs. @Value

The `@Value` annotation is a core container feature, and it does not provide the same features as type-safe configuration properties. The following table summarizes the features that are supported by `@ConfigurationProperties` and `@Value`:

Feature	<code>@ConfigurationProperties</code>	<code>@Value</code>
Relaxed binding	Yes	Limited (see note below)

Feature	@ConfigurationProperties	@Value
Meta-data support	Yes	No
SpEL evaluation	No	Yes

If you do want to use `@Value`, we recommend that you refer to property names using their canonical form (kebab-case using only lowercase letters). This will allow Spring Boot to use the same logic as it does when [relaxed binding @ConfigurationProperties](#).

NOTE

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

If you define a set of configuration keys for your own components, we recommend you group them in a POJO annotated with `@ConfigurationProperties`. Doing so will provide you with structured, type-safe object that you can inject into your own beans.

SpEL expressions from [application property files](#) are not processed at time of parsing these files and populating the environment. However, it is possible to write a SpEL expression in `@Value`. If the value of a property from an application property file is a SpEL expression, it will be evaluated when consumed through `@Value`.

7.3. Profiles

Spring Profiles provide a way to segregate parts of your application configuration and make it be available only in certain environments. Any `@Component`, `@Configuration` or `@ConfigurationProperties` can be marked with `@Profile` to limit when it is loaded, as shown in the following example:

Java

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

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

    // ...

}
```

Kotlin

```
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Profile

@Configuration(proxyBeanMethods = false)
@Profile("production")
class ProductionConfiguration {

    // ...

}
```

NOTE

If `@ConfigurationProperties` beans are registered through `@EnableConfigurationProperties` instead of automatic scanning, the `@Profile` annotation needs to be specified on the `@Configuration` class that has the `@EnableConfigurationProperties` annotation. In the case where `@ConfigurationProperties` are scanned, `@Profile` can be specified on the `@ConfigurationProperties` class itself.

You can use a `spring.profiles.active Environment` property to specify which profiles are active. You can specify the property in any of the ways described earlier in this chapter. For example, you could include it in your `application.properties`, as shown in the following example:

Properties

```
spring.profiles.active=dev,hsqldb
```

Yaml

```
spring:
  profiles:
    active: "dev,hsqldb"
```

You could also specify it on the command line by using the following switch: `--spring.profiles.active=dev,hsqldb`.

If no profile is active, a default profile is enabled. The name of the default profile is `default` and it can be tuned using the `spring.profiles.default Environment` property, as shown in the following example:

Properties

```
spring.profiles.default=none
```

Yaml

```
spring:  
  profiles:  
    default: "none"
```

`spring.profiles.active` and `spring.profiles.default` can only be used in non-profile-specific documents. This means they cannot be included in [profile specific files](#) or [documents activated by `spring.config.activate.on-profile`](#).

For example, the second document configuration is invalid:

Properties

```
# this document is valid  
spring.profiles.active=prod  
---  
# this document is invalid  
spring.config.activate.on-profile=prod  
spring.profiles.active=metrics
```

Yaml

```
# this document is valid  
spring:  
  profiles:  
    active: "prod"  
---  
# this document is invalid  
spring:  
  config:  
    activate:  
      on-profile: "prod"  
  profiles:  
    active: "metrics"
```

7.3.1. Adding Active Profiles

The `spring.profiles.active` property follows the same ordering rules as other properties: The highest `PropertySource` wins. This means that you can specify active profiles in `application.properties` and then **replace** them by using the command line switch.

Sometimes, it is useful to have properties that **add** to the active profiles rather than replace them. The `spring.profiles.include` property can be used to add active profiles on top of those activated by the `spring.profiles.active` property. The `SpringApplication` entry point also has a Java API for setting additional profiles. See the `setAdditionalProfiles()` method in `SpringApplication`.

For example, when an application with the following properties is run, the common and local profiles will be activated even when it runs using the `--spring.profiles.active` switch:

Properties

```
spring.profiles.include[0]=common  
spring.profiles.include[1]=local
```

Yaml

```
spring:  
  profiles:  
    include:  
      - "common"  
      - "local"
```

WARNING Similar to `spring.profiles.active`, `spring.profiles.include` can only be used in non-profile-specific documents. This means it cannot be included in [profile specific files](#) or [documents activated](#) by `spring.config.activate.on-profile`.

Profile groups, which are described in the [next section](#) can also be used to add active profiles if a given profile is active.

7.3.2. Profile Groups

Occasionally the profiles that you define and use in your application are too fine-grained and become cumbersome to use. For example, you might have `proddb` and `prodmq` profiles that you use to enable database and messaging features independently.

To help with this, Spring Boot lets you define profile groups. A profile group allows you to define a logical name for a related group of profiles.

For example, we can create a `production` group that consists of our `proddb` and `prodmq` profiles.

Properties

```
spring.profiles.group.production[0]=proddb  
spring.profiles.group.production[1]=prodmq
```

Yaml

```
spring:  
  profiles:  
    group:  
      production:  
        - "proddb"  
        - "prodmq"
```

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

WARNING

Similar to `spring.profiles.active` and `spring.profiles.include`, `spring.profiles.group` can only be used in non-profile-specific documents. This means it cannot be included in [profile specific files](#) or [documents activated by spring.config.activate.on-profile](#).

7.3.3. Programmatically Setting Profiles

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

7.3.4. Profile-specific Configuration Files

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

7.4. Logging

Spring Boot uses [Commons Logging](#) for all internal logging but leaves the underlying log implementation open. Default configurations are provided for [Java Util Logging](#), [Log4j2](#), and [Logback](#). In each case, loggers are pre-configured to use console output with optional file output also available.

By default, if you use the “Starters”, Logback is used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J all work correctly.

TIP

There are a lot of logging frameworks available for Java. Do not worry if the above list seems confusing. Generally, you do not need to change your logging dependencies and the Spring Boot defaults work just fine.

TIP

When you deploy your application to a servlet container or application server, logging performed with the Java Util Logging API is not routed into your application's logs. This prevents logging performed by the container or other applications that have been deployed to it from appearing in your application's logs.

7.4.1. Log Format

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

```

2024-06-20T08:04:21.437Z INFO 111727 --- [myapp] [main]
o.s.b.d.f.logexample.MyApplication : Starting MyApplication using Java 17.0.11
with PID 111727 (/opt/apps/myapp.jar started by myuser in /opt/apps/)
2024-06-20T08:04:21.456Z INFO 111727 --- [myapp] [main]
o.s.b.d.f.logexample.MyApplication : No active profile set, falling back to 1
default profile: "default"
2024-06-20T08:04:25.212Z INFO 111727 --- [myapp] [main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-06-20T08:04:25.253Z INFO 111727 --- [myapp] [main]
o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-06-20T08:04:25.253Z INFO 111727 --- [myapp] [main]
o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache
Tomcat/10.1.25]
2024-06-20T08:04:25.399Z INFO 111727 --- [myapp] [main]
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded
WebApplicationContext
2024-06-20T08:04:25.406Z INFO 111727 --- [myapp] [main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization
completed in 3723 ms
2024-06-20T08:04:26.611Z INFO 111727 --- [myapp] [main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with
context path ''
2024-06-20T08:04:26.640Z INFO 111727 --- [myapp] [main]
o.s.b.d.f.logexample.MyApplication : Started MyApplication in 6.738 seconds
(process running for 7.682)

```

The following items are output:

- Date and Time: Millisecond precision and easily sortable.
- Log Level: **ERROR**, **WARN**, **INFO**, **DEBUG**, or **TRACE**.
- Process ID.
- A **---** separator to distinguish the start of actual log messages.
- Application name: Enclosed in square brackets (logged by default only if `spring.application.name` is set)
- Thread name: Enclosed in square brackets (may be truncated for console output).
- Correlation ID: If tracing is enabled (not shown in the sample above)
- Logger name: This is usually the source class name (often abbreviated).
- The log message.

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

TIP If you have a `spring.application.name` property but don't want it logged you can set `logging.include-application-name` to `false`.

7.4.2. Console Output

The default log configuration echoes messages to the console as they are written. By default, **ERROR**-level, **WARN**-level, and **INFO**-level messages are logged. You can also enable a “debug” mode by starting your application with a **--debug** flag.

```
$ java -jar myapp.jar --debug
```

NOTE You can also specify **debug=true** in your **application.properties**.

When the debug mode is enabled, a selection of core loggers (embedded container, Hibernate, and Spring Boot) are configured to output more information. Enabling the debug mode does *not* configure your application to log all messages with **DEBUG** level.

Alternatively, you can enable a “trace” mode by starting your application with a **--trace** flag (or **trace=true** in your **application.properties**). Doing so enables trace logging for a selection of core loggers (embedded container, Hibernate schema generation, and the whole Spring portfolio).

Color-coded Output

If your terminal supports ANSI, color output is used to aid readability. You can set **spring.output.ansi.enabled** to a supported value to override the auto-detection.

Color coding is configured by using the **%clr** conversion word. In its simplest form, the converter colors the output according to the log level, as shown in the following example:

```
%clr(%5p)
```

The following table describes the mapping of log levels to colors:

Level	Color
FATAL	Red
ERROR	Red
WARN	Yellow
INFO	Green
DEBUG	Green
TRACE	Green

Alternatively, you can specify the color or style that should be used by providing it as an option to the conversion. For example, to make the text yellow, use the following setting:

```
%clr(%d{yyyy-MM-dd'T'HH:mm:ss.SSSXXX}){yellow}
```

The following colors and styles are supported:

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

7.4.3. File Output

By default, Spring Boot logs only to the console and does not write log files. If you want to write log files in addition to the console output, you need to set a `logging.file.name` or `logging.file.path` property (for example, in your `application.properties`).

The following table shows how the `logging.*` properties can be used together:

Table 8. Logging properties

<code>logging.file.name</code>	<code>logging.file.path</code>	Example	Description
(<i>none</i>)	(<i>none</i>)		Console only logging.
Specific file	(<i>none</i>)	<code>my.log</code>	Writes to the specified log file. Names can be an exact location or relative to the current directory.
(<i>none</i>)	Specific directory	<code>/var/log</code>	Writes <code>spring.log</code> to the specified directory. Names can be an exact location or relative to the current directory.

Log files rotate when they reach 10 MB and, as with console output, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged by default.

TIP Logging properties are independent of the actual logging infrastructure. As a result, specific configuration keys (such as `logback.configurationFile` for Logback) are not managed by Spring Boot.

7.4.4. File Rotation

If you are using the Logback, it is possible to fine-tune log rotation settings using your `application.properties` or `application.yaml` file. For all other logging system, you will need to configure rotation settings directly yourself (for example, if you use Log4j2 then you could add a `log4j2.xml` or `log4j2-spring.xml` file).

The following rotation policy properties are supported:

Name	Description
<code>logging.logback.rollingpolicy.file-name-pattern</code>	The filename pattern used to create log archives.

Name	Description
<code>logging.logback.rollingpolicy.clean-history-on-start</code>	If log archive cleanup should occur when the application starts.
<code>logging.logback.rollingpolicy.max-file-size</code>	The maximum size of log file before it is archived.
<code>logging.logback.rollingpolicy.total-size-cap</code>	The maximum amount of size log archives can take before being deleted.
<code>logging.logback.rollingpolicy.max-history</code>	The maximum number of archive log files to keep (defaults to 7).

7.4.5. Log Levels

All the supported logging systems can have the logger levels set in the Spring Environment (for example, in `application.properties`) by using `logging.level.<logger-name>=<level>` where `level` is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, or OFF. The `root` logger can be configured by using `logging.level.root`.

The following example shows potential logging settings in `application.properties`:

Properties

```
logging.level.root=warn
logging.level.org.springframework.web=debug
logging.level.org.hibernate=error
```

Yaml

```
logging:
  level:
    root: "warn"
    org.springframework.web: "debug"
    org.hibernate: "error"
```

It is also possible to set logging levels using environment variables. For example, `LOGGING_LEVEL_ORG_SPRINGFRAMEWORK_WEB=DEBUG` will set `org.springframework.web` to `DEBUG`.

NOTE

The above approach will only work for package level logging. Since relaxed binding always converts environment variables to lowercase, it is not possible to configure logging for an individual class in this way. If you need to configure logging for a class, you can use the `SPRING_APPLICATION_JSON` variable.

7.4.6. Log Groups

It is often useful to be able to group related loggers together so that they can all be configured at the same time. For example, you might commonly change the logging levels for *all* Tomcat related loggers, but you can not easily remember top level packages.

To help with this, Spring Boot allows you to define logging groups in your Spring [Environment](#). For example, here is how you could define a “tomcat” group by adding it to your `application.properties`:

Properties

```
logging.group.tomcat=org.apache.catalina,org.apache.coyote,org.apache.tomcat
```

Yaml

```
logging:
  group:
    tomcat: "org.apache.catalina,org.apache.coyote,org.apache.tomcat"
```

Once defined, you can change the level for all the loggers in the group with a single line:

Properties

```
logging.level.tomcat=trace
```

Yaml

```
logging:
  level:
    tomcat: "trace"
```

Spring Boot includes the following pre-defined logging groups that can be used out-of-the-box:

Name	Loggers
web	<code>org.springframework.core.codec, org.springframework.http,</code> <code>org.springframework.web, org.springframework.boot.actuate.endpoint.web,</code> <code>org.springframework.boot.web.servlet.ServletContextInitializerBeans</code>
sql	<code>org.springframework.jdbc.core, org.hibernate.SQL,</code> <code>org.jooq.tools.LoggerListener</code>

7.4.7. Using a Log Shutdown Hook

In order to release logging resources when your application terminates, a shutdown hook that will trigger log system cleanup when the JVM exits is provided. This shutdown hook is registered automatically unless your application is deployed as a war file. If your application has complex context hierarchies the shutdown hook may not meet your needs. If it does not, disable the shutdown hook and investigate the options provided directly by the underlying logging system. For example, Logback offers [context selectors](#) which allow each Logger to be created in its own context. You can use the `logging.register-shutdown-hook` property to disable the shutdown hook. Setting it to `false` will disable the registration. You can set the property in your `application.properties` or `application.yaml` file:

Properties

```
logging.register-shutdown-hook=false
```

Yaml

```
logging:  
  register-shutdown-hook: false
```

7.4.8. Custom Log Configuration

The various logging systems can be activated by including the appropriate libraries on the classpath and can be further customized by providing a suitable configuration file in the root of the classpath or in a location specified by the following Spring [Environment](#) property: `logging.config`.

You can force Spring Boot to use a particular logging system by using the `org.springframework.boot.logging.LoggingSystem` system property. The value should be the fully qualified class name of a [LoggingSystem](#) implementation. You can also disable Spring Boot's logging configuration entirely by using a value of `none`.

NOTE Since logging is initialized **before** the [ApplicationContext](#) is created, it is not possible to control logging from [@PropertySources](#) in Spring [@Configuration](#) files. The only way to change the logging system or disable it entirely is through System properties.

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

Logging System	Customization
Logback	<code>logback-spring.xml</code> , <code>logback-spring.groovy</code> , <code>logback.xml</code> , or <code>logback.groovy</code>
Log4j2	<code>log4j2-spring.xml</code> or <code>log4j2.xml</code>
JDK (Java Util Logging)	<code>logging.properties</code>

NOTE When possible, we recommend that you use the `-spring` variants for your logging configuration (for example, `logback-spring.xml` rather than `logback.xml`). If you use standard configuration locations, Spring cannot completely control log initialization.

WARNING There are known classloading issues with Java Util Logging that cause problems when running from an 'executable jar'. We recommend that you avoid it when running from an 'executable jar' if at all possible.

To help with the customization, some other properties are transferred from the Spring [Environment](#) to System properties. This allows the properties to be consumed by logging system configuration. For example, setting `logging.file.name` in `application.properties` or `LOGGING_FILE_NAME` as an environment variable will result in the `LOG_FILE` System property being set. The properties that are transferred are described in the following table:

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

If you use Logback, the following properties are also transferred:

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

Spring Environment	System Property	Comments
<code>logging.logback.rollingpolicy.max-history</code>	<code>LOGBACK_ROLLINGPOLICY_MAX_HISTORY</code>	Maximum number of archive log files to keep.

All the supported logging systems can consult System properties when parsing their configuration files. See the default configurations in `spring-boot.jar` for examples:

- [Logback](#)
- [Log4j 2](#)
- [Java Util logging](#)

TIP If you want to use a placeholder in a logging property, you should use [Spring Boot's syntax](#) and not the syntax of the underlying framework. Notably, if you use Logback, you should use `:` as the delimiter between a property name and its default value and not use `:-`.

You can add MDC and other ad-hoc content to log lines by overriding only the `LOG_LEVEL_PATTERN` (or `logging.pattern.level` with Logback). For example, if you use `logging.pattern.level=user:%X{user} %5p`, then the default log format contains an MDC entry for "user", if it exists, as shown in the following example.

TIP

```
2019-08-30 12:30:04.031 user:someone INFO 22174 --- [nio-8080-exec-0]
demo.Controller
Handling authenticated request
```

7.4.9. Logback Extensions

Spring Boot includes a number of extensions to Logback that can help with advanced configuration. You can use these extensions in your `logback-spring.xml` configuration file.

NOTE Because the standard `logback.xml` configuration file is loaded too early, you cannot use extensions in it. You need to either use `logback-spring.xml` or define a `logging.config` property.

WARNING The extensions cannot be used with Logback's [configuration scanning](#). If you attempt to do so, making changes to the configuration file results in an error similar to one of the following being logged:

```
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for
[springProperty], current ElementPath is [[configuration][springProperty]]
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for
[springProfile], current ElementPath is [[configuration][springProfile]]
```

Profile-specific Configuration

The `<springProfile>` tag lets you optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the `<configuration>` element. Use the `name` attribute to specify which profile accepts the configuration. The `<springProfile>` tag can contain a profile name (for example `staging`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example `production & (eu-central | eu-west)`. Check the [Spring Framework reference guide](#) for more details. The following listing shows three sample profiles:

```
<springProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
</springProfile>

<springProfile name="dev | staging">
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active
-->
</springProfile>

<springProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is not active -->
</springProfile>
```

Environment Properties

The `<springProperty>` tag lets you expose properties from the Spring [Environment](#) for use within Logback. Doing so can be useful if you want to access values from your `application.properties` file in your Logback configuration. The tag works in a similar way to Logback's standard `<property>` tag. However, rather than specifying a direct `value`, you specify the `source` of the property (from the [Environment](#)). If you need to store the property somewhere other than in `local` scope, you can use the `scope` attribute. If you need a fallback value (in case the property is not set in the [Environment](#)), you can use the `defaultValue` attribute. The following example shows how to expose properties for use within Logback:

```
<springProperty scope="context" name="fluentHost" source="myapp.fluentd.host"
    defaultValue="localhost"/>
<appender name="FLUENT" class="ch.qos.logback.more.appenders.DataFluentAppender">
    <remoteHost>${fluentHost}</remoteHost>
    ...
</appender>
```

NOTE

The `source` must be specified in kebab case (such as `my.property-name`). However, properties can be added to the [Environment](#) by using the relaxed rules.

7.4.10. Log4j2 Extensions

Spring Boot includes a number of extensions to Log4j2 that can help with advanced configuration.

You can use these extensions in any `log4j2-spring.xml` configuration file.

- NOTE** Because the standard `log4j2.xml` configuration file is loaded too early, you cannot use extensions in it. You need to either use `log4j2-spring.xml` or define a `logging.config` property.
- NOTE** The extensions supersede the [Spring Boot support](#) provided by Log4J. You should make sure not to include the `org.apache.logging.log4j:log4j-spring-boot` module in your build.

Profile-specific Configuration

The `<SpringProfile>` tag lets you optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the `<Configuration>` element. Use the `name` attribute to specify which profile accepts the configuration. The `<SpringProfile>` tag can contain a profile name (for example `staging`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed, for example `production & (eu-central | eu-west)`. Check the [Spring Framework reference guide](#) for more details. The following listing shows three sample profiles:

```
<SpringProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
</SpringProfile>

<SpringProfile name="dev | staging">
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active
-->
</SpringProfile>

<SpringProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is not active -->
</SpringProfile>
```

Environment Properties Lookup

If you want to refer to properties from your Spring [Environment](#) within your Log4j2 configuration you can use `spring:` prefixed [lookups](#). Doing so can be useful if you want to access values from your `application.properties` file in your Log4j2 configuration.

The following example shows how to set a Log4j2 property named `applicationName` that reads `spring.application.name` from the Spring [Environment](#):

```
<Properties>
    <Property name="applicationName">${spring:spring.application.name}</Property>
</Properties>
```

NOTE The lookup key should be specified in kebab case (such as `my.property-name`).

Log4j2 System Properties

Log4j2 supports a number of [System Properties](#) that can be used to configure various items. For example, the `log4j2.skipJansi` system property can be used to configure if the [ConsoleAppender](#) will try to use a [Jansi](#) output stream on Windows.

All system properties that are loaded after the Log4j2 initialization can be obtained from the Spring [Environment](#). For example, you could add `log4j2.skipJansi=false` to your [application.properties](#) file to have the [ConsoleAppender](#) use Jansi on Windows.

NOTE The Spring [Environment](#) is only considered when system properties and OS environment variables do not contain the value being loaded.

WARNING System properties that are loaded during early Log4j2 initialization cannot reference the Spring [Environment](#). For example, the property Log4j2 uses to allow the default Log4j2 implementation to be chosen is used before the Spring Environment is available.

7.5. Internationalization

Spring Boot supports localized messages so that your application can cater to users of different language preferences. By default, Spring Boot looks for the presence of a [messages](#) resource bundle at the root of the classpath.

NOTE The auto-configuration applies when the default properties file for the configured resource bundle is available (`messages.properties` by default). If your resource bundle contains only language-specific properties files, you are required to add the default. If no properties file is found that matches any of the configured base names, there will be no auto-configured [MessageSource](#).

The basename of the resource bundle as well as several other attributes can be configured using the `spring.messages` namespace, as shown in the following example:

Properties

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

Yaml

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

TIP

`spring.messages.basename` supports comma-separated list of locations, either a package qualifier or a resource resolved from the classpath root.

See [MessageSourceProperties](#) for more supported options.

7.6. Aspect-Oriented Programming

Spring Boot provides auto-configuration for aspect-oriented programming (AOP). You can learn more about AOP with Spring in the [Spring Framework reference documentation](#).

By default, Spring Boot's auto-configuration configures Spring AOP to use CGLib proxies. To use JDK proxies instead, set `configprop:spring.aop.proxy-target-class` to `false`.

If AspectJ is on the classpath, Spring Boot's auto-configuration will automatically enable AspectJ auto proxy such that `@EnableAspectJAutoProxy` is not required.

7.7. JSON

Spring Boot provides integration with three JSON mapping libraries:

- Gson
- Jackson
- JSON-B

Jackson is the preferred and default library.

7.7.1. Jackson

Auto-configuration for Jackson is provided and Jackson is part of `spring-boot-starter-json`. When Jackson is on the classpath an `ObjectMapper` bean is automatically configured. Several configuration properties are provided for [customizing the configuration of the ObjectMapper](#).

Custom Serializers and Deserializers

If you use Jackson to serialize and deserialize JSON data, you might want to write your own `JsonSerializer` and `JsonDeserializer` classes. Custom serializers are usually [registered with Jackson through a module](#), but Spring Boot provides an alternative `@JsonComponent` annotation that makes it easier to directly register Spring Beans.

You can use the `@JsonComponent` annotation directly on `JsonSerializer`, `JsonDeserializer` or `KeyDeserializer` implementations. You can also use it on classes that contain serializers/deserializers as inner classes, as shown in the following example:

Java

```
import java.io.IOException;

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.ObjectCodec;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonDeserializer;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.JsonSerializer;
import com.fasterxml.jackson.databind.SerializerProvider;

import org.springframework.boot.jackson.JsonComponent;

@JsonComponent
public class MyJsonComponent {

    public static class Serializer extends JsonSerializer<MyObject> {

        @Override
        public void serialize(MyObject value, JsonGenerator jgen, SerializerProvider
serializers) throws IOException {
            jgen.writeStartObject();
            jgen.writeStringField("name", value.getName());
            jgen.writeNumberField("age", value.getAge());
            jgen.writeEndObject();
        }
    }

    public static class Deserializer extends JsonDeserializer<MyObject> {

        @Override
        public MyObject deserialize(JsonParser jsonParser, DeserializationContext
ctxt) throws IOException {
            ObjectCodec codec = jsonParser.getCodec();
            JsonNode tree = codec.readTree(jsonParser);
            String name = tree.get("name").textValue();
            int age = tree.get("age").intValue();
            return new MyObject(name, age);
        }
    }
}
```

```

import com.fasterxml.jackson.core.JsonGenerator
import com.fasterxml.jackson.core.JsonParser
import com.fasterxml.jackson.core.JsonProcessingException
import com.fasterxml.jackson.databind.DeserializationContext
import com.fasterxml.jackson.databind.JsonDeserializer
import com.fasterxml.jackson.databind.JsonNode
import com.fasterxml.jackson.databind.JsonSerializer
import com.fasterxml.jackson.databind.SerializerProvider
import org.springframework.boot.jackson.JsonComponent
import java.io.IOException

@JsonComponent
class MyJsonComponent {

    class Serializer : JsonSerializer<MyObject>() {
        @Throws(IOException::class)
        override fun serialize(value: MyObject, jgen: JsonGenerator, serializers: SerializerProvider) {
            jgen.writeStartObject()
            jgen.writeStringField("name", value.name)
            jgen.writeNumberField("age", value.age)
            jgen.writeEndObject()
        }
    }

    class Deserializer : JsonDeserializer<MyObject>() {
        @Throws(IOException::class, JsonProcessingException::class)
        override fun deserialize(jsonParser: JsonParser, ctxt: DeserializationContext): MyObject {
            val codec = jsonParser.codec
            val tree = codec.readTree<JsonNode>(jsonParser)
            val name = tree["name"].textValue()
            val age = tree["age"].intValue()
            return MyObject(name, age)
        }
    }
}

```

All `@JsonComponent` beans in the `ApplicationContext` are automatically registered with Jackson. Because `@JsonComponent` is meta-annotated with `@Component`, the usual component-scanning rules apply.

Spring Boot also provides `JsonObjectSerializer` and `JsonObjectDeserializer` base classes that provide useful alternatives to the standard Jackson versions when serializing objects. See `JsonObjectSerializer` and `JsonObjectDeserializer` in the Javadoc for details.

The example above can be rewritten to use `JsonObjectSerializer/JsonObjectDeserializer` as follows:

Java

```
import java.io.IOException;

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.ObjectCodec;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.SerializerProvider;

import org.springframework.boot.jackson.JsonComponent;
import org.springframework.boot.jackson.JsonObjectDeserializer;
import org.springframework.boot.jackson.JsonObjectSerializer;

@JsonComponent
public class MyJsonComponent {

    public static class Serializer extends JsonObjectSerializer<MyObject> {

        @Override
        protected void serializeObject(MyObject value, JsonGenerator jgen,
        SerializerProvider provider)
            throws IOException {
            jgen.writeStringField("name", value.getName());
            jgen.writeNumberField("age", value.getAge());
        }
    }

    public static class Deserializer extends JsonObjectDeserializer<MyObject> {

        @Override
        protected MyObject deserializeObject(JsonParser jsonParser,
        DeserializationContext context, ObjectCodec codec,
            JsonNode tree) throws IOException {
            String name = nullSafeValue(tree.get("name"), String.class);
            int age = nullSafeValue(tree.get("age"), Integer.class);
            return new MyObject(name, age);
        }
    }
}
```

```

`object`



import com.fasterxml.jackson.core.JsonGenerator
import com.fasterxml.jackson.core.JsonParser
import com.fasterxml.jackson.core.ObjectCodec
import com.fasterxml.jackson.databind.DeserializationContext
import com.fasterxml.jackson.databind.JsonNode
import com.fasterxml.jackson.databind.SerializerProvider
import org.springframework.boot.jackson.JsonComponent
import org.springframework.boot.jackson.JsonObjectDeserializer
import org.springframework.boot.jackson.JsonObjectSerializer
import java.io.IOException


@JsonComponent
class MyJsonComponent {

    class Serializer : JsonObjectSerializer<MyObject>() {
        @Throws(IOException::class)
        override fun serializeObject(value: MyObject, jgen: JsonGenerator, provider: SerializerProvider) {
            jgen.writeStringField("name", value.name)
            jgen.writeNumberField("age", value.age)
        }
    }

    class Deserializer : JsonObjectDeserializer<MyObject>() {
        @Throws(IOException::class)
        override fun deserializeObject(jsonParser: JsonParser, context: DeserializationContext,
                                      codec: ObjectCodec, tree: JsonNode): MyObject {
            val name = nullSafeValue(tree["name"], String::class.java)
            val age = nullSafeValue(tree["age"], Int::class.java)
            return MyObject(name, age)
        }
    }

}

```

Mixins

Jackson has support for mixins that can be used to mix additional annotations into those already declared on a target class. Spring Boot's Jackson auto-configuration will scan your application's packages for classes annotated with `@JsonMixin` and register them with the auto-configured `ObjectMapper`. The registration is performed by Spring Boot's `JsonMixinModule`.

7.7.2. Gson

Auto-configuration for Gson is provided. When Gson is on the classpath a `Gson` bean is automatically

configured. Several `spring.gson.*` configuration properties are provided for customizing the configuration. To take more control, one or more `GsonBuilderCustomizer` beans can be used.

7.7.3. JSON-B

Auto-configuration for JSON-B is provided. When the JSON-B API and an implementation are on the classpath a `Jsonb` bean will be automatically configured. The preferred JSON-B implementation is Eclipse Yasson for which dependency management is provided.

7.8. Task Execution and Scheduling

In the absence of an `Executor` bean in the context, Spring Boot auto-configures an `AsyncTaskExecutor`. When virtual threads are enabled (using Java 21+ and `spring.threads.virtual.enabled` set to `true`) this will be a `SimpleAsyncTaskExecutor` that uses virtual threads. Otherwise, it will be a `ThreadPoolTaskExecutor` with sensible defaults. In either case, the auto-configured executor will be automatically used for:

- asynchronous task execution (`@EnableAsync`)
- Spring for GraphQL's asynchronous handling of `Callable` return values from controller methods
- Spring MVC's asynchronous request processing
- Spring WebFlux's blocking execution support

If you have defined a custom `Executor` in the context, both regular task execution (that is `@EnableAsync`) and Spring for GraphQL will use it. However, the Spring MVC and Spring WebFlux support will only use it if it is an `AsyncTaskExecutor` implementation (named `applicationTaskExecutor`). Depending on your target arrangement, you could change your `Executor` into an `AsyncTaskExecutor` or define both an `AsyncTaskExecutor` and an `AsyncConfigurer` wrapping your custom `Executor`.

TIP

The auto-configured `ThreadPoolTaskExecutorBuilder` allows you to easily create instances that reproduce what the auto-configuration does by default.

When a `ThreadPoolTaskExecutor` is auto-configured, the thread pool uses 8 core threads that can grow and shrink according to the load. Those default settings can be fine-tuned using the `spring.task.execution` namespace, as shown in the following example:

Properties

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

Yaml

```
spring:
  task:
    execution:
      pool:
        max-size: 16
        queue-capacity: 100
        keep-alive: "10s"
```

This changes the thread pool to use a bounded queue so that when the queue is full (100 tasks), the thread pool increases to maximum 16 threads. Shrinking of the pool is more aggressive as threads are reclaimed when they are idle for 10 seconds (rather than 60 seconds by default).

A scheduler can also be auto-configured if it needs to be associated with scheduled task execution (using `@EnableScheduling` for instance).

If virtual threads are enabled (using Java 21+ and `spring.threads.virtual.enabled` set to `true`) this will be a `SimpleAsyncTaskScheduler` that uses virtual threads. This `SimpleAsyncTaskScheduler` will ignore any pooling related properties.

If virtual threads are not enabled, it will be a `ThreadPoolTaskScheduler` with sensible defaults. The `ThreadPoolTaskScheduler` uses one thread by default and its settings can be fine-tuned using the `spring.task.scheduling` namespace, as shown in the following example:

Properties

```
spring.task.scheduling.thread-name-prefix=scheduling-
spring.task.scheduling.pool.size=2
```

Yaml

```
spring:
  task:
    scheduling:
      thread-name-prefix: "scheduling-"
      pool:
        size: 2
```

A `ThreadPoolTaskExecutorBuilder` bean, a `SimpleAsyncTaskExecutorBuilder` bean, a `ThreadPoolTaskSchedulerBuilder` bean and a `SimpleAsyncTaskSchedulerBuilder` are made available in the context if a custom executor or scheduler needs to be created. The `SimpleAsyncTaskExecutorBuilder` and `SimpleAsyncTaskSchedulerBuilder` beans are auto-configured to use virtual threads if they are enabled (using Java 21+ and `spring.threads.virtual.enabled` set to `true`).

7.9. Testing

Spring Boot provides a number of utilities and annotations to help when testing your application. Test support is provided by two modules: `spring-boot-test` contains core items, and `spring-boot-test-autoconfigure` supports auto-configuration for tests.

Most developers use the `spring-boot-starter-test` “Starter”, which imports both Spring Boot test modules as well as JUnit Jupiter, AssertJ, Hamcrest, and a number of other useful libraries.

If you have tests that use JUnit 4, JUnit 5’s vintage engine can be used to run them. To use the vintage engine, add a dependency on `junit-vintage-engine`, as shown in the following example:

TIP

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

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

7.9.1. Test Scope Dependencies

The `spring-boot-starter-test` “Starter” (in the `test` scope) contains the following provided libraries:

- [JUnit 5](#): The de-facto standard for unit testing Java applications.
- [Spring Test](#) & [Spring Boot Test](#): Utilities and integration test support for Spring Boot applications.
- [AssertJ](#): A fluent assertion library.
- [Hamcrest](#): A library of matcher objects (also known as constraints or predicates).
- [Mockito](#): A Java mocking framework.
- [JSONassert](#): An assertion library for JSON.
- [JsonPath](#): XPath for JSON.
- [Awaitility](#): A library for testing asynchronous systems.

We generally find these common libraries to be useful when writing tests. If these libraries do not suit your needs, you can add additional test dependencies of your own.

7.9.2. Testing Spring Applications

One of the major advantages of dependency injection is that it should make your code easier to unit test. You can instantiate objects by using the `new` operator without even involving Spring. You can also use *mock objects* instead of real dependencies.

Often, you need to move beyond unit testing and start integration testing (with a Spring `ApplicationContext`). It is useful to be able to perform integration testing without requiring deployment of your application or needing to connect to other infrastructure.

The Spring Framework includes a dedicated test module for such integration testing. You can declare a dependency directly to `org.springframework:spring-test` or use the `spring-boot-starter-test` “Starter” to pull it in transitively.

If you have not used the `spring-test` module before, you should start by reading the [relevant section](#) of the Spring Framework reference documentation.

7.9.3. Testing Spring Boot Applications

A Spring Boot application is a Spring `ApplicationContext`, so nothing very special has to be done to test it beyond what you would normally do with a vanilla Spring context.

NOTE

External properties, logging, and other features of Spring Boot are installed in the context by default only if you use `SpringApplication` to create it.

Spring Boot provides a `@SpringBootTest` annotation, which can be used as an alternative to the standard `spring-test @ContextConfiguration` annotation when you need Spring Boot features. The annotation works by [creating the ApplicationContext used in your tests through SpringApplication](#). In addition to `@SpringBootTest` a number of other annotations are also provided for [testing more specific slices](#) of an application.

TIP

If you are using JUnit 4, do not forget to also add `@RunWith(SpringRunner.class)` to your test, otherwise the annotations will be ignored. If you are using JUnit 5, there is no need to add the equivalent `@ExtendWith(SpringExtension.class)` as `@SpringBootTest` and the other `@...Test` annotations are already annotated with it.

By default, `@SpringBootTest` will not start a server. You can use the `webEnvironment` attribute of `@SpringBootTest` to further refine how your tests run:

- **MOCK(Default)** : Loads a web `ApplicationContext` and provides a mock web environment. Embedded servers are not started when using this annotation. If a web environment is not available on your classpath, this mode transparently falls back to creating a regular non-web `ApplicationContext`. It can be used in conjunction with `@AutoConfigureMockMvc` or `@AutoConfigureWebTestClient` for mock-based testing of your web application.
- **RANDOM_PORT**: Loads a `WebServerApplicationContext` and provides a real web environment. Embedded servers are started and listen on a random port.
- **DEFINED_PORT**: Loads a `WebServerApplicationContext` and provides a real web environment. Embedded servers are started and listen on a defined port (from your `application.properties`)

or on the default port of `8080`.

- **NONE:** Loads an `ApplicationContext` by using `SpringApplication` but does not provide *any* web environment (mock or otherwise).

NOTE If your test is `@Transactional`, it rolls back the transaction at the end of each test method by default. However, as using this arrangement with either `RANDOM_PORT` or `DEFINED_PORT` implicitly provides a real servlet environment, the HTTP client and server run in separate threads and, thus, in separate transactions. Any transaction initiated on the server does not roll back in this case.

NOTE `@SpringBootTest` with `webEnvironment = WebEnvironment.RANDOM_PORT` will also start the management server on a separate random port if your application uses a different port for the management server.

Detecting Web Application Type

If Spring MVC is available, a regular MVC-based application context is configured. If you have only Spring WebFlux, we will detect that and configure a WebFlux-based application context instead.

If both are present, Spring MVC takes precedence. If you want to test a reactive web application in this scenario, you must set the `spring.main.web-application-type` property:

Java

```
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(properties = "spring.main.web-application-type=reactive")
class MyWebFluxTests {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.context.SpringBootTest

@SpringBootTest(properties = ["spring.main.web-application-type=reactive"])
class MyWebFluxTests {

    // ...

}
```

Detecting Test Configuration

If you are familiar with the Spring Test Framework, you may be used to using `@ContextConfiguration(classes=…)` in order to specify which Spring `@Configuration` to load.

Alternatively, you might have often used nested `@Configuration` classes within your test.

When testing Spring Boot applications, this is often not required. Spring Boot's `@*Test` annotations search for your primary configuration automatically whenever you do not explicitly define one.

The search algorithm works up from the package that contains the test until it finds a class annotated with `@SpringBootApplication` or `@SpringBootConfiguration`. As long as you [structured your code](#) in a sensible way, your main configuration is usually found.

If you use a [test annotation to test a more specific slice of your application](#), you should avoid adding configuration settings that are specific to a particular area on the [main method's application class](#).

NOTE The underlying component scan configuration of `@SpringBootApplication` defines exclude filters that are used to make sure slicing works as expected. If you are using an explicit `@ComponentScan` directive on your `@SpringBootApplication`-annotated class, be aware that those filters will be disabled. If you are using slicing, you should define them again.

If you want to customize the primary configuration, you can use a nested `@TestConfiguration` class. Unlike a nested `@Configuration` class, which would be used instead of your application's primary configuration, a nested `@TestConfiguration` class is used in addition to your application's primary configuration.

NOTE Spring's test framework caches application contexts between tests. Therefore, as long as your tests share the same configuration (no matter how it is discovered), the potentially time-consuming process of loading the context happens only once.

Using the Test Configuration Main Method

Typically the test configuration discovered by `@SpringBootTest` will be your main `@SpringBootApplication`. In most well structured applications, this configuration class will also include the `main` method used to launch the application.

For example, the following is a very common code pattern for a typical Spring Boot application:

Java

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

@SpringBootApplication
public class MyApplication {

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

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import
org.springframework.boot.docs.using.structuringyourcode.locatingthemainclass.MyApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

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

In the example above, the `main` method doesn't do anything other than delegate to `SpringApplication.run`. It is, however, possible to have a more complex `main` method that applies customizations before calling `SpringApplication.run`.

For example, here is an application that changes the banner mode and sets additional profiles:

Java

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

@SpringBootApplication
public class MyApplication {

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

Kotlin

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

@SpringBootApplication
class MyApplication

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

Since customizations in the `main` method can affect the resulting `ApplicationContext`, it's possible that you might also want to use the `main` method to create the `ApplicationContext` used in your tests. By default, `@SpringBootTest` will not call your `main` method, and instead the class itself is used directly to create the `ApplicationContext`.

If you want to change this behavior, you can change the `useMainMethod` attribute of `@SpringBootTest` to `UseMainMethod.ALWAYS` or `UseMainMethod.WHEN_AVAILABLE`. When set to `ALWAYS`, the test will fail if no `main` method can be found. When set to `WHEN_AVAILABLE` the `main` method will be used if it is available, otherwise the standard loading mechanism will be used.

For example, the following test will invoke the `main` method of `MyApplication` in order to create the `ApplicationContext`. If the main method sets additional profiles then those will be active when the `ApplicationContext` starts.

Java

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

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.UseMainMethod;

@SpringBootTest(useMainMethod = UseMainMethod.ALWAYS)
class MyApplicationTests {

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

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.UseMainMethod

@SpringBootTest(useMainMethod = UseMainMethod.ALWAYS)
class MyApplicationTests {

    @Test
    fun exampleTest() {
        // ...
    }

}
```

Excluding Test Configuration

If your application uses component scanning (for example, if you use `@SpringBootApplication` or `@ComponentScan`), you may find top-level configuration classes that you created only for specific tests accidentally get picked up everywhere.

As we [have seen earlier](#), `@TestConfiguration` can be used on an inner class of a test to customize the primary configuration. `@TestConfiguration` can also be used on a top-level class. Doing so indicates that the class should not be picked up by scanning. You can then import the class explicitly where it is required, as shown in the following example:

Java

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

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Import;

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

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

}
```

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.context.annotation.Import

@SpringBootTest
@Import(MyTestsConfiguration::class)
class MyTests {

    @Test
    fun exampleTest() {
        // ...
    }

}
```

NOTE

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

NOTE

An imported `@TestConfiguration` is processed earlier than an inner-class `@TestConfiguration` and an imported `@TestConfiguration` will be processed before any configuration found through component scanning. Generally speaking, this difference in ordering has no noticeable effect but it is something to be aware of if you're relying on bean overriding.

Using Application Arguments

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

Java

```
import org.junit.jupiter.api.Test;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.ApplicationArguments;  
import org.springframework.boot.test.context.SpringBootTest;  
  
import static org.assertj.core.api.Assertions.assertThat;  
  
@SpringBootTest(args = "--app.test=one")  
class MyApplicationArgumentTests {  
  
    @Test  
    void applicationArgumentsPopulated(@Autowired ApplicationArguments args) {  
        assertThat(args.getOptionNames()).containsOnly("app.test");  
        assertThat(args.getOptionValues("app.test")).containsOnly("one");  
    }  
  
}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat  
import org.junit.jupiter.api.Test  
import org.springframework.beans.factory.annotation.Autowired  
import org.springframework.boot.ApplicationArguments  
import org.springframework.boot.test.context.SpringBootTest  
  
@SpringBootTest(args = ["--app.test=one"])  
class MyApplicationArgumentTests {  
  
    @Test  
    fun applicationArgumentsPopulated(@Autowired args: ApplicationArguments) {  
        assertThat(args.optionNames).containsOnly("app.test")  
        assertThat(args.getOptionValues("app.test")).containsOnly("one")  
    }  
  
}
```

Testing With a Mock Environment

By default, `@SpringBootTest` does not start the server but instead sets up a mock environment for testing web endpoints.

With Spring MVC, we can query our web endpoints using `MockMvc` or `WebTestClient`, as shown in the following example:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.reactive.server.WebTestClient;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
class MyMockMvcTests {

    @Test
    void testWithMockMvc(@Autowired MockMvc mvc) throws Exception {

        mvc.perform(get("/")).andExpect(status().isOk()).andExpect(content().string("Hello
World"));
    }

    // If Spring WebFlux is on the classpath, you can drive MVC tests with a
    // WebTestClient
    @Test
    void testWithWebTestClient(@Autowired WebTestClient webClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello World");
    }
}
```

```

import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
import org.springframework.test.web.servlet.result.MockMvcResultMatchers

@SpringBootTest
@AutoConfigureMockMvc
class MyMockMvcTests {

    @Test
    fun testWithMockMvc(@Autowired mvc: MockMvc) {

        mvc.perform(MockMvcRequestBuilders.get("/")).andExpect(MockMvcResultMatchers.status().isOk)
            .andExpect(MockMvcResultMatchers.content().string("Hello World"))
    }

    // If Spring WebFlux is on the classpath, you can drive MVC tests with a
    // WebTestClient

    @Test
    fun testWithWebTestClient(@Autowired webClient: WebTestClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Hello World")
    }
}

```

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

With Spring WebFlux endpoints, you can use `WebTestClient` as shown in the following example:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestClient;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.reactive.server.WebTestClient;

@SpringBootTest
@AutoConfigureWebTestClient
class MyMockWebTestClientTests {

    @Test
    void exampleTest(@Autowired WebTestClient webClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello World");
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestClient
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody

@SpringBootTest
@AutoConfigureWebTestClient
class MyMockWebTestClientTests {

    @Test
    fun exampleTest(@Autowired webClient: WebTestClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Hello World")
    }

}
```

Testing within a mocked environment is usually faster than running with a full servlet container. However, since mocking occurs at the Spring MVC layer, code that relies on lower-level servlet container behavior cannot be directly tested with MockMvc.

- TIP** For example, Spring Boot’s error handling is based on the “error page” support provided by the servlet container. This means that, whilst you can test your MVC layer throws and handles exceptions as expected, you cannot directly test that a specific [custom error page](#) is rendered. If you need to test these lower-level concerns, you can start a fully running server as described in the next section.

Testing With a Running Server

If you need to start a full running server, we recommend that you use random ports. If you use `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`, an available port is picked at random each time your test runs.

The `@LocalServerPort` annotation can be used to [inject the actual port used](#) into your test. For convenience, tests that need to make REST calls to the started server can additionally `@Autowired` a `WebTestClient`, which resolves relative links to the running server and comes with a dedicated API for verifying responses, as shown in the following example:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.test.web.reactive.server.WebTestClient;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortWebTestClientTests {

    @Test
    void exampleTest(@Autowired WebTestClient webClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello World");
    }

}
```

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortWebTestClientTests {

    @Test
    fun exampleTest(@Autowired webClient: WebTestClient) {
        webClient
            .get().uri("/")
            .exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Hello World")
    }

}
```

TIP `WebTestClient` can also be used with a [mock environment](#), removing the need for a running server, by annotating your test class with `@AutoConfigureWebTestClient`.

This setup requires `spring-webflux` on the classpath. If you can not or will not add webflux, Spring Boot also provides a `TestRestTemplate` facility:

Java

```
import org.junit.jupiter.api.Test;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;  
import org.springframework.boot.test.web.client.TestRestTemplate;  
  
import static org.assertj.core.api.Assertions.assertThat;  
  
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)  
class MyRandomPortTestRestTemplateTests {  
  
    @Test  
    void exampleTest(@Autowired TestRestTemplate restTemplate) {  
        String body = restTemplate.getForObject("/", String.class);  
        assertThat(body).isEqualTo("Hello World");  
    }  
  
}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat  
import org.junit.jupiter.api.Test  
import org.springframework.beans.factory.annotation.Autowired  
import org.springframework.boot.test.context.SpringBootTest  
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment  
import org.springframework.boot.test.web.client.TestRestTemplate  
  
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)  
class MyRandomPortTestRestTemplateTests {  
  
    @Test  
    fun exampleTest(@Autowired restTemplate: TestRestTemplate) {  
        val body = restTemplate.getForObject("/", String::class.java)  
        assertThat(body).isEqualTo("Hello World")  
    }  
  
}
```

Customizing WebTestClient

To customize the `WebTestClient` bean, configure a `WebTestClientBuilderCustomizer` bean. Any such beans are called with the `WebTestClient.Builder` that is used to create the `WebTestClient`.

Using JMX

As the test context framework caches context, JMX is disabled by default to prevent identical components to register on the same domain. If such test needs access to an [MBeanServer](#), consider marking it dirty as well:

Java

```
import javax.management.MBeanServer;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.DirtiesContext;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(properties = "spring.jmx.enabled=true")
@DirtiesContext
class MyJmxTests {

    @Autowired
    private MBeanServer mBeanServer;

    @Test
    void exampleTest() {
        assertThat(this.mBeanServer.getDomains()).contains("java.lang");
        // ...
    }
}
```

```
import javax.management.MBeanServer

import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.annotation.DirtiesContext

@SpringBootTest(properties = ["spring.jmx.enabled=true"])
@DirtiesContext
class MyJmxTests(@Autowired val mBeanServer: MBeanServer) {

    @Test
    fun exampleTest() {
        assertThat(mBeanServer.domains).contains("java.lang")
        // ...
    }
}
```

Using Observations

If you annotate a sliced test with `@AutoConfigureObservability`, it auto-configures an `ObservationRegistry`.

Using Metrics

Regardless of your classpath, meter registries, except the in-memory backed, are not auto-configured when using `@SpringBootTest`.

If you need to export metrics to a different backend as part of an integration test, annotate it with `@AutoConfigureObservability`.

If you annotate a sliced test with `@AutoConfigureObservability`, it auto-configures an in-memory `MeterRegistry`. Data exporting in sliced tests is not supported with the `@AutoConfigureObservability` annotation.

Using Tracing

Regardless of your classpath, tracing components which are reporting data are not auto-configured when using `@SpringBootTest`.

If you need those components as part of an integration test, annotate the test with `@AutoConfigureObservability`.

If you have created your own reporting components (e.g. a custom `SpanExporter` or `SpanHandler`) and you don't want them to be active in tests, you can use the `@ConditionalOnEnabledTracing` annotation to disable them.

If you annotate a sliced test with `@AutoConfigureObservability`, it auto-configures a no-op `Tracer`. Data exporting in sliced tests is not supported with the `@AutoConfigureObservability` annotation.

Mocking and Spying Beans

When running tests, it is sometimes necessary to mock certain components within your application context. For example, you may have a facade over some remote service that is unavailable during development. Mocking can also be useful when you want to simulate failures that might be hard to trigger in a real environment.

Spring Boot includes a `@MockBean` annotation that can be used to define a Mockito mock for a bean inside your `ApplicationContext`. You can use the annotation to add new beans or replace a single existing bean definition. The annotation can be used directly on test classes, on fields within your test, or on `@Configuration` classes and fields. When used on a field, the instance of the created mock is also injected. Mock beans are automatically reset after each test method.

If your test uses one of Spring Boot's test annotations (such as `@SpringBootTest`), this feature is automatically enabled. To use this feature with a different arrangement, listeners must be explicitly added, as shown in the following example:

Java

```
import org.springframework.boot.test.mock.mockito.MockitoTestExecutionListener;
import org.springframework.boot.test.mock.mockito.ResetMocksTestExecutionListener;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.TestExecutionListeners;

@ContextConfiguration(classes = MyConfig.class)
@TestExecutionListeners({ MockitoTestExecutionListener.class,
    ResetMocksTestExecutionListener.class })
class MyTests {

    // ...

}
```

NOTE

Kotlin

```
import org.springframework.boot.test.mock.mockito.MockitoTestExecutionListener
import org.springframework.boot.test.mock.mockito.ResetMocksTestExecutionListener
import org.springframework.test.context.ContextConfiguration
import org.springframework.test.context.TestExecutionListeners

@ContextConfiguration(classes = [MyConfig::class])
@TestExecutionListeners(
    MockitoTestExecutionListener::class,
    ResetMocksTestExecutionListener::class
)
class MyTests {

    // ...

}
```

The following example replaces an existing `RemoteService` bean with a mock implementation:

Java

```
import org.junit.jupiter.api.Test;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.boot.test.mock.mockito.MockBean;  
  
import static org.assertj.core.api.Assertions.assertThat;  
import static org.mockito.BDDMockito.given;  
  
@SpringBootTest  
class MyTests {  
  
    @Autowired  
    private Reverser reverser;  
  
    @MockBean  
    private RemoteService remoteService;  
  
    @Test  
    void exampleTest() {  
        given(this.remoteService.getValue()).willReturn("spring");  
        String reverse = this.reverser.getReverseValue(); // Calls injected  
        RemoteService  
        assertThat(reverse).isEqualTo("gnirps");  
    }  
}
```

```

import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.mock.mockito.MockBean

@SpringBootTest
class MyTests(@Autowired val reverser: Reverser, @MockBean val remoteService: RemoteService) {

    @Test
    fun exampleTest() {
        given(remoteService.value).willReturn("spring")
        val reverse = reverser.reverseValue // Calls injected RemoteService
        assertThat(reverse).isEqualTo("gnirps")
    }
}

```

NOTE

`@MockBean` cannot be used to mock the behavior of a bean that is exercised during application context refresh. By the time the test is executed, the application context refresh has completed and it is too late to configure the mocked behavior. We recommend using a `@Bean` method to create and configure the mock in this situation.

Additionally, you can use `@SpyBean` to wrap any existing bean with a Mockito `spy`. See the [Javadoc](#) for full details.

NOTE

While Spring's test framework caches application contexts between tests and reuses a context for tests sharing the same configuration, the use of `@MockBean` or `@SpyBean` influences the cache key, which will most likely increase the number of contexts.

TIP

If you are using `@SpyBean` to spy on a bean with `@Cacheable` methods that refer to parameters by name, your application must be compiled with `-parameters`. This ensures that the parameter names are available to the caching infrastructure once the bean has been spied upon.

TIP

When you are using `@SpyBean` to spy on a bean that is proxied by Spring, you may need to remove Spring's proxy in some situations, for example when setting expectations using `given` or `when`. Use `AopTestUtils.getTargetObject(yourProxiedSpy)` to do so.

Auto-configured Tests

Spring Boot's auto-configuration system works well for applications but can sometimes be a little too much for tests. It often helps to load only the parts of the configuration that are required to test a "slice" of your application. For example, you might want to test that Spring MVC controllers are

mapping URLs correctly, and you do not want to involve database calls in those tests, or you might want to test JPA entities, and you are not interested in the web layer when those tests run.

The `spring-boot-test-autoconfigure` module includes a number of annotations that can be used to automatically configure such “slices”. Each of them works in a similar way, providing a `@…Test` annotation that loads the `ApplicationContext` and one or more `@AutoConfigure…` annotations that can be used to customize auto-configuration settings.

NOTE

Each slice restricts component scan to appropriate components and loads a very restricted set of auto-configuration classes. If you need to exclude one of them, most `@…Test` annotations provide an `excludeAutoConfiguration` attribute. Alternatively, you can use `@ImportAutoConfiguration#exclude`.

NOTE

Including multiple “slices” by using several `@…Test` annotations in one test is not supported. If you need multiple “slices”, pick one of the `@…Test` annotations and include the `@AutoConfigure…` annotations of the other “slices” by hand.

TIP

It is also possible to use the `@AutoConfigure…` annotations with the standard `@SpringBootTest` annotation. You can use this combination if you are not interested in “slicing” your application but you want some of the auto-configured test beans.

Auto-configured JSON Tests

To test that object JSON serialization and deserialization is working as expected, you can use the `@JsonTest` annotation. `@JsonTest` auto-configures the available supported JSON mapper, which can be one of the following libraries:

- Jackson `ObjectMapper`, any `@JsonComponent` beans and any Jackson `Modules`
- `Gson`
- `Jsonb`

TIP

A list of the auto-configurations that are enabled by `@JsonTest` can be [found in the appendix](#).

If you need to configure elements of the auto-configuration, you can use the `@AutoConfigureJsonTesters` annotation.

Spring Boot includes AssertJ-based helpers that work with the `JSONAssert` and `JsonPath` libraries to check that JSON appears as expected. The `JacksonTester`, `GsonTester`, `JsonbTester`, and `BasicJsonTester` classes can be used for Jackson, Gson, Jsonb, and Strings respectively. Any helper fields on the test class can be `@Autowired` when using `@JsonTest`. The following example shows a test class for Jackson:

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.json.JsonTest;
import org.springframework.boot.test.json.JacksonTester;

import static org.assertj.core.api.Assertions.assertThat;

@JsonTest
class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    void serialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda", "Civic");
        // Assert against a '.json' file in the same package as the test
        assertThat(this.json.write(details)).isEqualToJson("expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");

        assertThat(this.json.write(details)).extractingJsonPathStringValue("@.make").isEqualTo
        ("Honda");
    }

    @Test
    void deserialize() throws Exception {
        String content = "{\"make\":\"Ford\", \"model\":\"Focus\"}";
        assertThat(this.json.parse(content)).isEqualTo(new VehicleDetails("Ford",
        "Focus"));
        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
    }
}
```

```

import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.json.JsonTest
import org.springframework.boot.test.json.JacksonTester

@JsonTest
class MyJsonTests(@Autowired val json: JacksonTester<VehicleDetails>) {

    @Test
    fun serialize() {
        val details = VehicleDetails("Honda", "Civic")
        // Assert against a '.json' file in the same package as the test
        assertThat(json.write(details)).isEqualToJson("expected.json")
        // Or use JSON path based assertions
        assertThat(json.write(details)).hasJsonPathStringValue("@.make")

        assertThat(json.write(details)).extractingJsonPathStringValue("@.make").isEqualTo("Hon
da")
    }

    @Test
    fun deserialize() {
        val content = "{\"make\":\"Ford\", \"model\":\"Focus\"}"
        assertThat(json.parse(content)).isEqualTo(VehicleDetails("Ford", "Focus"))
        assertThat(json.parseObject(content).make).isEqualTo("Ford")
    }
}

```

NOTE

JSON helper classes can also be used directly in standard unit tests. To do so, call the `initFields` method of the helper in your `@Before` method if you do not use `@JsonTest`.

If you use Spring Boot's AssertJ-based helpers to assert on a number value at a given JSON path, you might not be able to use `isEqualTo` depending on the type. Instead, you can use AssertJ's `satisfies` to assert that the value matches the given condition. For instance, the following example asserts that the actual number is a float value close to `0.15` within an offset of `0.01`.

Java

```
@Test
void someTest() throws Exception {
    SomeObject value = new SomeObject(0.152f);

    assertThat(this.json.write(value)).extractingJsonPathNumberValue("@.test.numberValue")
        .satisfies((number) -> assertThat(number.floatValue()).isCloseTo(0.15f,
within(0.01f)));
}
```

Kotlin

```
@Test
fun someTest() {
    val value = SomeObject(0.152f)
    assertThat(json.write(value)).extractingJsonPathNumberValue("@.test.numberValue")
        .satisfies(ThrowingConsumer { number ->
            assertThat(number.toFloat()).isCloseTo(0.15f, within(0.01f))
        })
}
```

Auto-configured Spring MVC Tests

To test whether Spring MVC controllers are working as expected, use the `@WebMvcTest` annotation. `@WebMvcTest` auto-configures the Spring MVC infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `Filter`, `HandlerInterceptor`, `WebMvcConfigurer`, `WebMvcRegistrations`, and `HandlerMethodArgumentResolver`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@WebMvcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP A list of the auto-configuration settings that are enabled by `@WebMvcTest` can be found [in the appendix](#).

TIP If you need to register extra components, such as the Jackson `Module`, you can import additional configuration classes by using `@Import` on your test.

Often, `@WebMvcTest` is limited to a single controller and is used in combination with `@MockBean` to provide mock implementations for required collaborators.

`@WebMvcTest` also auto-configures `MockMvc`. Mock MVC offers a powerful way to quickly test MVC controllers without needing to start a full HTTP server.

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

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import static org.mockito.BDDMockito.given;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(UserVehicleController.class)
class MyControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));
        this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk())
            .andExpect(content().string("Honda Civic"));
    }

}
```

```

import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.boot.test.mock.mockito.MockBean
import org.springframework.http.MediaType
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
import org.springframework.test.web.servlet.result.MockMvcResultMatchers

@WebMvcTest(UserVehicleController::class)
class MyControllerTests(@Autowired val mvc: MockMvc) {

    @MockBean
    lateinit var userVehicleService: UserVehicleService

    @Test
    fun testExample() {
        given(userVehicleService.getVehicleDetails("sboot"))
            .willReturn(VehicleDetails("Honda", "Civic"))

        mvc.perform(MockMvcRequestBuilders.get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
            .andExpect(MockMvcResultMatchers.status().isOk)
            .andExpect(MockMvcResultMatchers.content().string("Honda Civic"))
    }
}

```

TIP If you need to configure elements of the auto-configuration (for example, when servlet filters should be applied) you can use attributes in the `@AutoConfigureMockMvc` annotation.

If you use HtmlUnit and Selenium, auto-configuration also provides an HtmlUnit `WebClient` bean and/or a Selenium `WebDriver` bean. The following example uses HtmlUnit:

Java

```
import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.HtmlPage;
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.given;

@WebMvcTest(UserVehicleController.class)
class MyHtmlUnitTests {

    @Autowired
    private WebClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot")).willReturn(new
VehicleDetails("Honda", "Civic"));
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
        assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
    }

}
```

```
import com.gargoylesoftware.htmlunit.WebClient
import com.gargoylesoftware.htmlunit.html.HtmlPage
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.boot.test.mock.mockito.MockBean

@WebMvcTest(UserVehicleController::class)
class MyHtmlUnitTests(@Autowired val webClient: WebClient) {

    @MockBean
    lateinit var userVehicleService: UserVehicleService

    @Test
    fun testExample() {

        given(userVehicleService.getVehicleDetails("sboot")).willReturn(VehicleDetails("Honda"
        , "Civic"))
            val page = webClient.getPage<HtmlPage>("/sboot/vehicle.html")
            assertThat(page.body.textContent).isEqualTo("Honda Civic")
    }

}
```

NOTE By default, Spring Boot puts `WebDriver` beans in a special “scope” to ensure that the driver exits after each test and that a new instance is injected. If you do not want this behavior, you can add `@Scope("singleton")` to your `WebDriver @Bean` definition.

WARNING The `webDriver` scope created by Spring Boot will replace any user defined scope of the same name. If you define your own `webDriver` scope you may find it stops working when you use `@WebMvcTest`.

If you have Spring Security on the classpath, `@WebMvcTest` will also scan `WebSecurityConfigurer` beans. Instead of disabling security completely for such tests, you can use Spring Security’s test support. More details on how to use Spring Security’s `MockMvc` support can be found in this [Testing With Spring Security](#) how-to section.

TIP Sometimes writing Spring MVC tests is not enough; Spring Boot can help you run [full end-to-end tests with an actual server](#).

Auto-configured Spring WebFlux Tests

To test that `Spring WebFlux` controllers are working as expected, you can use the `@WebFluxTest` annotation. `@WebFluxTest` auto-configures the Spring WebFlux infrastructure and limits scanned

beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `WebFilter`, and `WebFluxConfigurer`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@WebFluxTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

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

TIP If you need to register extra components, such as Jackson `Module`, you can import additional configuration classes using `@Import` on your test.

Often, `@WebFluxTest` is limited to a single controller and used in combination with the `@MockBean` annotation to provide mock implementations for required collaborators.

`@WebFluxTest` also auto-configures `WebTestClient`, which offers a powerful way to quickly test WebFlux controllers without needing to start a full HTTP server.

TIP You can also auto-configure `WebTestClient` in a non-`@WebFluxTest` (such as `@SpringBootTest`) by annotating it with `@AutoConfigureWebTestClient`. The following example shows a class that uses both `@WebFluxTest` and a `WebTestClient`:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.reactive.server.WebTestClient;

import static org.mockito.BDDMockito.given;

@WebFluxTest(UserVehicleController.class)
class MyControllerTests {

    @Autowired
    private WebTestClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    void testExample() {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));

        this.webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN).exchange()
            .expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Honda Civic");
    }

}
```

```

import org.junit.jupiter.api.Test
import org.mockito.BDDMockito.given
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest
import org.springframework.boot.test.mock.mockito.MockBean
import org.springframework.http.MediaType
import org.springframework.test.web.reactive.server.WebTestClient
import org.springframework.test.web.reactive.server.expectBody

@WebFluxTest(UserVehicleController::class)
class MyControllerTests(@Autowired val webClient: WebTestClient) {

    @MockBean
    lateinit var userVehicleService: UserVehicleService

    @Test
    fun testExample() {
        given(userVehicleService.getVehicleDetails("sboot"))
            .willReturn(VehicleDetails("Honda", "Civic"))
        webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN).exchange()
            .expectStatus().isOk
            .expectBody<String>().isEqualTo("Honda Civic")
    }
}

```

TIP This setup is only supported by WebFlux applications as using `WebTestClient` in a mocked web application only works with WebFlux at the moment.

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

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

TIP Sometimes writing Spring WebFlux tests is not enough; Spring Boot can help you run [full end-to-end tests with an actual server](#).

Auto-configured Spring GraphQL Tests

Spring GraphQL offers a dedicated testing support module; you'll need to add it to your project:

Maven

```
<dependencies>
  <dependency>
    <groupId>org.springframework.graphql</groupId>
    <artifactId>spring-graphql-test</artifactId>
    <scope>test</scope>
  </dependency>
  <!-- Unless already present in the compile scope -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Gradle

```
dependencies {
  testImplementation("org.springframework.graphql:spring-graphql-test")
  // Unless already present in the implementation configuration
  testImplementation("org.springframework.boot:spring-boot-starter-webflux")
}
```

This testing module ships the [GraphQLTester](#). The tester is heavily used in test, so be sure to become familiar with using it. There are [GraphQLTester](#) variants and Spring Boot will auto-configure them depending on the type of tests:

- the [ExecutionGraphQLServiceTester](#) performs tests on the server side, without a client nor a transport
- the [HttpGraphQLTester](#) performs tests with a client that connects to a server, with or without a live server

Spring Boot helps you to test your [Spring GraphQL Controllers](#) with the [@GraphQLTest](#) annotation. [@GraphQLTest](#) auto-configures the Spring GraphQL infrastructure, without any transport nor server being involved. This limits scanned beans to [@Controller](#), [RuntimeWiringConfigurer](#), [JsonComponent](#), [Converter](#), [GenericConverter](#), [DataFetcherExceptionResolver](#), [Instrumentation](#) and [GraphQLSourceBuilderCustomizer](#). Regular [@Component](#) and [@ConfigurationProperties](#) beans are not scanned when the [@GraphQLTest](#) annotation is used. [@EnableConfigurationProperties](#) can be used to include [@ConfigurationProperties](#) beans.

TIP A list of the auto-configurations that are enabled by [@GraphQLTest](#) can be [found in the appendix](#).

Often, [@GraphQLTest](#) is limited to a set of controllers and used in combination with the [@MockBean](#) annotation to provide mock implementations for required collaborators.

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.docs.web.graphql.runtimewiring.GreetingController;
import org.springframework.boot.test.autoconfigure.graphql.GraphQLTest;
import org.springframework.graphql.test.tester.GraphQLTester;

@GraphQLTest(GreetingController.class)
class GreetingControllerTests {

    @Autowired
    private GraphQLTester graphQLTester;

    @Test
    void shouldGreetWithSpecificName() {
        this.graphQLTester.document("{ greeting(name: \"Alice\") } ")
            .execute()
            .path("greeting")
            .entity(String.class)
            .isEqualTo("Hello, Alice!");
    }

    @Test
    void shouldGreetWithDefaultName() {
        this.graphQLTester.document("{ greeting } ")
            .execute()
            .path("greeting")
            .entity(String.class)
            .isEqualTo("Hello, Spring!");
    }

}
```

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.docs.web.graphql.runtimewiring.GreetingController
import org.springframework.boot.test.autoconfigure.graphql.GraphQLTest
import org.springframework.graphql.test.tester.GraphQLTester

@GraphQLTest(GreetingController::class)
internal class GreetingControllerTests {

    @Autowired
    lateinit var graphQLTester: GraphQLTester

    @Test
    fun shouldGreetWithSpecificName() {
        graphQLTester.document("{ greeting(name: \"Alice\") }")
            .execute().path("greeting").entity(String::class.java)
            .isEqualTo("Hello, Alice!")
    }

    @Test
    fun shouldGreetWithDefaultName() {
        graphQLTester.document("{ greeting }")
            .execute().path("greeting").entity(String::class.java)
            .isEqualTo("Hello, Spring!")
    }

}
```

@SpringBootTest tests are full integration tests and involve the entire application. When using a random or defined port, a live server is configured and an **HttpGraphQLTester** bean is contributed automatically so you can use it to test your server. When a MOCK environment is configured, you can also request an **HttpGraphQLTester** bean by annotating your test class with **@AutoConfigureHttpGraphQLTester**:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.graphql.tester.AutoConfigureHttpGraphQLTes
ter;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.graphql.test.tester.HttpGraphQLTester;

@AutoConfigureHttpGraphQLTester
@SpringBootTest(webEnvironment = SpringApplication.WebEnvironment.MOCK)
class GraphQLIntegrationTests {

    @Test
    void shouldGreetWithSpecificName(@Autowired HttpGraphQLTester graphQLTester) {
        HttpGraphQLTester authenticatedTester = graphQLTester.mutate()
            .webTestClient((client) -> client.defaultHeaders((headers) ->
headers.setBasicAuth("admin", "ilovespring")))
            .build();
        authenticatedTester.document("{ greeting(name: \"Alice\") } ")
            .execute()
            .path("greeting")
            .entity(String.class)
            .isEqualTo("Hello, Alice!");
    }

}
```

```

import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.graphql.tester.AutoConfigureHttpGraphQLTester
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.graphql.test.tester.HttpGraphQLTester
import org.springframework.http.HttpHeaders
import org.springframework.test.web.reactive.server.WebTestClient

@AutoConfigureHttpGraphQLTester
@SpringBootTest(webEnvironment = SpringApplication.WebEnvironment.MOCK)
class GraphQLIntegrationTests {

    @Test
    fun shouldGreetWithSpecificName(@Autowired graphQLTester: HttpGraphQLTester) {
        val authenticatedTester = graphQLTester.mutate()
            .webTestClient { client: WebTestClient.Builder ->
                client.defaultHeaders { headers: HttpHeaders ->
                    headers.setBasicAuth("admin", "ilovespring")
                }
            }.build()
        authenticatedTester.document("{ greeting(name: \"Alice\") } ").execute()
            .path("greeting").entity(String::class.java).isEqualTo("Hello, Alice!")
    }
}

```

Auto-configured Data Cassandra Tests

You can use `@DataCassandraTest` to test Cassandra applications. By default, it configures a `CassandraTemplate`, scans for `@Table` classes, and configures Spring Data Cassandra repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataCassandraTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Cassandra with Spring Boot, see "[Cassandra](#)".)

TIP A list of the auto-configuration settings that are enabled by `@DataCassandraTest` can be [found in the appendix](#).

The following example shows a typical setup for using Cassandra tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.cassandra.DataCassandraTest;

@DataCassandraTest
class MyDataCassandraTests {

    @Autowired
    private SomeRepository repository;

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.cassandra.DataCassandraTest

@DataCassandraTest
class MyDataCassandraTests(@Autowired val repository: SomeRepository)
```

Auto-configured Data Couchbase Tests

You can use `@DataCouchbaseTest` to test Couchbase applications. By default, it configures a `CouchbaseTemplate` or `ReactiveCouchbaseTemplate`, scans for `@Document` classes, and configures Spring Data Couchbase repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataCouchbaseTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Couchbase with Spring Boot, see "[Couchbase](#)", earlier in this chapter.)

TIP A list of the auto-configuration settings that are enabled by `@DataCouchbaseTest` can be [found in the appendix](#).

The following example shows a typical setup for using Couchbase tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.couchbase.DataCouchbaseTest;

@DataCouchbaseTest
class MyDataCouchbaseTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.couchbase.DataCouchbaseTest

@DataCouchbaseTest
class MyDataCouchbaseTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

Auto-configured Data Elasticsearch Tests

You can use `@DataElasticsearchTest` to test Elasticsearch applications. By default, it configures an `ElasticsearchRestTemplate`, scans for `@Document` classes, and configures Spring Data Elasticsearch repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataElasticsearchTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Elasticsearch with Spring Boot, see "Elasticsearch", earlier in this chapter.)



A list of the auto-configuration settings that are enabled by `@DataElasticsearchTest` can be [found in the appendix](#).

The following example shows a typical setup for using Elasticsearch tests in Spring Boot:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.data.elasticsearch.DataElasticsearchTest;

@DataElasticsearchTest
class MyDataElasticsearchTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.data.elasticsearch.DataElasticsearchTest

@DataElasticsearchTest
class MyDataElasticsearchTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

Auto-configured Data JPA Tests

You can use the `@DataJpaTest` annotation to test JPA applications. By default, it scans for `@Entity` classes and configures Spring Data JPA repositories. If an embedded database is available on the classpath, it configures one as well. SQL queries are logged by default by setting the `spring.jpa.show-sql` property to `true`. This can be disabled using the `showSql` attribute of the annotation.

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



A list of the auto-configuration settings that are enabled by `@DataJpaTest` can be found [in the appendix](#).

By default, data JPA tests are transactional and roll back at the end of each test. See the [relevant section](#) in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class as follows:

Java

```
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyNonTransactionalTests {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest
import org.springframework.transaction.annotation.Propagation
import org.springframework.transaction.annotation.Transactional

@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyNonTransactionalTests {

    // ...

}
```

Data JPA tests may also inject a `TestEntityManager` bean, which provides an alternative to the standard JPA `EntityManager` that is specifically designed for tests.

TIP

`TestEntityManager` can also be auto-configured to any of your Spring-based test class by adding `@AutoConfigureTestEntityManager`. When doing so, make sure that your test is running in a transaction, for instance by adding `@Transactional` on your test class or method.

A `JdbcTemplate` is also available if you need that. The following example shows the `@DataJpaTest` annotation in use:

Java

```
import org.junit.jupiter.api.Test;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;  
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;  
  
import static org.assertj.core.api.Assertions.assertThat;  
  
@DataJpaTest  
class MyRepositoryTests {  
  
    @Autowired  
    private TestEntityManager entityManager;  
  
    @Autowired  
    private UserRepository repository;  
  
    @Test  
    void testExample() {  
        this.entityManager.persist(new User("sboot", "1234"));  
        User user = this.repository.findByUsername("sboot");  
        assertThat(user.getUsername()).isEqualTo("sboot");  
        assertThat(user.getEmployeeNumber()).isEqualTo("1234");  
    }  
}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat  
import org.junit.jupiter.api.Test  
import org.springframework.beans.factory.annotation.Autowired  
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest  
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager  
  
@DataJpaTest  
class MyRepositoryTests(@Autowired val entityManager: TestEntityManager, @Autowired  
val repository: UserRepository) {  
  
    @Test  
    fun testExample() {  
        entityManager.persist(User("sboot", "1234"))  
        val user = repository.findByUsername("sboot")  
        assertThat(user?.username).isEqualTo("sboot")  
        assertThat(user?.employeeNumber).isEqualTo("1234")  
    }  
}
```

In-memory embedded databases generally work well for tests, since they are fast and do not require any installation. If, however, you prefer to run tests against a real database you can use the `@AutoConfigureTestDatabase` annotation, as shown in the following example:

Java

```
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase.Replace;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
class MyRepositoryTests {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest

@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class MyRepositoryTests {

    // ...

}
```

Auto-configured JDBC Tests

`@JdbcTest` is similar to `@DataJpaTest` but is for tests that only require a `DataSource` and do not use Spring Data JDBC. By default, it configures an in-memory embedded database and a `JdbcTemplate`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@JdbcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP

A list of the auto-configurations that are enabled by `@JdbcTest` can be [found in the appendix](#).

By default, JDBC tests are transactional and roll back at the end of each test. See the [relevant section](#) in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class, as follows:

Java

```
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyTransactionalTests {

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest
import org.springframework.transaction.annotation.Propagation
import org.springframework.transaction.annotation.Transactional

@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyTransactionalTests
```

If you prefer your test to run against a real database, you can use the [@AutoConfigureTestDatabase](#) annotation in the same way as for [@DataJpaTest](#). (See "Auto-configured Data JPA Tests".)

Auto-configured Data JDBC Tests

[@DataJdbcTest](#) is similar to [@JdbcTest](#) but is for tests that use Spring Data JDBC repositories. By default, it configures an in-memory embedded database, a [JdbcTemplate](#), and Spring Data JDBC repositories. Only [AbstractJdbcConfiguration](#) subclasses are scanned when the [@DataJdbcTest](#) annotation is used, regular [@Component](#) and [@ConfigurationProperties](#) beans are not scanned. [@EnableConfigurationProperties](#) can be used to include [@ConfigurationProperties](#) beans.

TIP

A list of the auto-configurations that are enabled by [@DataJdbcTest](#) can be [found in the appendix](#).

By default, Data JDBC tests are transactional and roll back at the end of each test. See the [relevant section](#) in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole test class as [shown in the JDBC example](#).

If you prefer your test to run against a real database, you can use the [@AutoConfigureTestDatabase](#) annotation in the same way as for [@DataJpaTest](#). (See "Auto-configured Data JPA Tests".)

Auto-configured Data R2DBC Tests

[@DataR2dbcTest](#) is similar to [@DataJdbcTest](#) but is for tests that use Spring Data R2DBC repositories. By default, it configures an in-memory embedded database, an [R2dbcEntityTemplate](#), and Spring Data R2DBC repositories. Regular [@Component](#) and [@ConfigurationProperties](#) beans are not scanned

when the `@DataR2dbcTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP A list of the auto-configurations that are enabled by `@DataR2dbcTest` can be [found in the appendix](#).

By default, Data R2DBC tests are not transactional.

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `@DataJpaTest`. (See "Auto-configured Data JPA Tests".)

Auto-configured jOOQ Tests

You can use `@JooqTest` in a similar fashion as `@JdbcTest` but for jOOQ-related tests. As jOOQ relies heavily on a Java-based schema that corresponds with the database schema, the existing `DataSource` is used. If you want to replace it with an in-memory database, you can use `@AutoConfigureTestDatabase` to override those settings. (For more about using jOOQ with Spring Boot, see "Using jOOQ".) Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@JooqTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP A list of the auto-configurations that are enabled by `@JooqTest` can be [found in the appendix](#).

`@JooqTest` configures a `DSLContext`. The following example shows the `@JooqTest` annotation in use:

Java

```
import org.jooq.DSLContext;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;

@JooqTest
class MyJooqTests {

    @Autowired
    private DSLContext dslContext;

    // ...

}
```

```
import org.jooq.DSLContext
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.jooq.JooqTest

@JooqTest
class MyJooqTests(@Autowired val dslContext: DSLContext) {

    // ...

}
```

JOOQ tests are transactional and roll back at the end of each test by default. If that is not what you want, you can disable transaction management for a test or for the whole test class as [shown in the JDBC example](#).

Auto-configured Data MongoDB Tests

You can use `@DataMongoTest` to test MongoDB applications. By default, it configures a `MongoTemplate`, scans for `@Document` classes, and configures Spring Data MongoDB repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataMongoTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using MongoDB with Spring Boot, see "[MongoDB](#)".)

TIP A list of the auto-configuration settings that are enabled by `@DataMongoTest` can be [found in the appendix](#).

The following class shows the `@DataMongoTest` annotation in use:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.data.mongodb.core.MongoTemplate;

@DataMongoTest
class MyDataMongoDbTests {

    @Autowired
    private MongoTemplate mongoTemplate;

    // ...

}
```

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.mongo.DataMongoTest
import org.springframework.data.mongodb.core.MongoTemplate

@DataMongoTest
class MyDataMongoDbTests(@Autowired val mongoTemplate: MongoTemplate) {

    // ...

}
```

Auto-configured Data Neo4j Tests

You can use `@DataNeo4jTest` to test Neo4j applications. By default, it scans for `@Node` classes, and configures Spring Data Neo4j repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataNeo4jTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Neo4J with Spring Boot, see ["Neo4j"](#).)

TIP A list of the auto-configuration settings that are enabled by `@DataNeo4jTest` can be [found in the appendix](#).

The following example shows a typical setup for using Neo4J tests in Spring Boot:

Java

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

@DataNeo4jTest
class MyDataNeo4jTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

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

@DataNeo4jTest
class MyDataNeo4jTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

By default, Data Neo4j tests are transactional and roll back at the end of each test. See the [relevant section](#) in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class, as follows:

Java

```
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyDataNeo4jTests {

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest
import org.springframework.transaction.annotation.Propagation
import org.springframework.transaction.annotation.Transactional

@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyDataNeo4jTests
```

NOTE

Transactional tests are not supported with reactive access. If you are using this style, you must configure `@DataNeo4jTest` tests as described above.

Auto-configured Data Redis Tests

You can use `@DataRedisTest` to test Redis applications. By default, it scans for `@RedisHash` classes and configures Spring Data Redis repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataRedisTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using Redis with Spring Boot, see "Redis".)

TIP

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

The following example shows the `@DataRedisTest` annotation in use:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest;

@DataRedisTest
class MyDataRedisTests {

    @Autowired
    private SomeRepository repository;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest

@DataRedisTest
class MyDataRedisTests(@Autowired val repository: SomeRepository) {

    // ...

}
```

Auto-configured Data LDAP Tests

You can use `@DataLdapTest` to test LDAP applications. By default, it configures an in-memory embedded LDAP (if available), configures an `LdapTemplate`, scans for `@Entry` classes, and configures Spring Data LDAP repositories. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@DataLdapTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans. (For more about using LDAP with Spring Boot, see "LDAP".)

TIP

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

The following example shows the `@DataLdapTest` annotation in use:

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.ldap.core.LdapTemplate;

@DataLdapTest
class MyDataLdapTests {

    @Autowired
    private LdapTemplate ldapTemplate;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest
import org.springframework.ldap.core.LdapTemplate

@DataLdapTest
class MyDataLdapTests(@Autowired val ldapTemplate: LdapTemplate) {

    // ...

}
```

In-memory embedded LDAP generally works well for tests, since it is fast and does not require any developer installation. If, however, you prefer to run tests against a real LDAP server, you should exclude the embedded LDAP auto-configuration, as shown in the following example:

Java

```
import
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;

@DataLdapTest(excludeAutoConfiguration = EmbeddedLdapAutoConfiguration.class)
class MyDataLdapTests {

    // ...

}
```

```
import  
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration  
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest  
  
@DataLdapTest(excludeAutoConfiguration = [EmbeddedLdapAutoConfiguration::class])  
class MyDataLdapTests {  
  
    // ...  
  
}
```

Auto-configured REST Clients

You can use the `@RestClientTest` annotation to test REST clients. By default, it auto-configures Jackson, GSON, and Jsonb support, configures a `RestTemplateBuilder` and a `RestClient.Builder`, and adds support for `MockRestServiceServer`. Regular `@Component` and `@ConfigurationProperties` beans are not scanned when the `@RestClientTest` annotation is used. `@EnableConfigurationProperties` can be used to include `@ConfigurationProperties` beans.

TIP A list of the auto-configuration settings that are enabled by `@RestClientTest` can be [found in the appendix](#).

The specific beans that you want to test should be specified by using the `value` or `components` attribute of `@RestClientTest`.

When using a `RestTemplateBuilder` in the beans under test and `RestTemplateBuilder.rootUri(String rootUri)` has been called when building the `RestTemplate`, then the root URI should be omitted from the `MockRestServiceServer` expectations as shown in the following example:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.client.RestClientTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.client.MockRestServiceServer;

import static org.assertj.core.api.Assertions.assertThat;
import static
org.springframework.test.web.client.match.MockRestRequestMatchers.requestTo;
import static
org.springframework.test.web.client.response.MockRestResponseCreators.withSuccess;

@RestClientTest(RemoteVehicleDetailsService.class)
class MyRestTemplateServiceTests {

    @Autowired
    private RemoteVehicleDetailsService service;

    @Autowired
    private MockRestServiceServer server;

    @Test
    void getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() {

        this.server.expect(requestTo("/greet/details")).andRespond(withSuccess("hello",
        MediaType.TEXT_PLAIN));
            String greeting = this.service.callRestService();
            assertThat(greeting).isEqualTo("hello");
    }

}
```

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.client.RestClientTest
import org.springframework.http.MediaType
import org.springframework.test.web.client.MockRestServiceServer
import org.springframework.test.web.client.match.MockRestRequestMatchers
import org.springframework.test.web.client.response.MockRestResponseCreators

@RestClientTest(RemoteVehicleDetailsService::class)
class MyRestTemplateServiceTests(
    @Autowired val service: RemoteVehicleDetailsService,
    @Autowired val server: MockRestServiceServer) {

    @Test
    fun getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() {
        server.expect(MockRestRequestMatchers.requestTo("/greet/details"))
            .andRespond(MockRestResponseCreators.withSuccess("hello",
        MediaType.TEXT_PLAIN))
        val greeting = service.callRestService()
        assertThat(greeting).isEqualTo("hello")
    }
}
```

When using a `RestClient.Builder` in the beans under test, or when using a `RestTemplateBuilder` without calling `rootUri(String rootURI)`, the full URI must be used in the `MockRestServiceServer` expectations as shown in the following example:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.client.RestClientTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.client.MockRestServiceServer;

import static org.assertj.core.api.Assertions.assertThat;
import static
org.springframework.test.web.client.match.MockRestRequestMatchers.requestTo;
import static
org.springframework.test.web.client.response.MockRestResponseCreators.withSuccess;

@RestClientTest(RemoteVehicleDetailsService.class)
class MyRestClientServiceTests {

    @Autowired
    private RemoteVehicleDetailsService service;

    @Autowired
    private MockRestServiceServer server;

    @Test
    void getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() {
        this.server.expect(requestTo("https://example.com/greet/details"))
            .andRespond(withSuccess("hello", MediaType.TEXT_PLAIN));
        String greeting = this.service.callRestService();
        assertThat(greeting).isEqualTo("hello");
    }

}
```

```

import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.web.client.RestClientTest
import org.springframework.http.MediaType
import org.springframework.test.web.client.MockRestServiceServer
import org.springframework.test.web.client.match.MockRestRequestMatchers
import org.springframework.test.web.client.response.MockRestResponseCreators

@RestClientTest(RemoteVehicleDetailsService::class)
class MyRestClientServiceTests(
    @Autowired val service: RemoteVehicleDetailsService,
    @Autowired val server: MockRestServiceServer) {

    @Test
    fun getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() {

        server.expect(MockRestRequestMatchers.requestTo("https://example.com/greet/details"))
            .andRespond(MockRestResponseCreators.withSuccess("hello",
        MediaType.TEXT_PLAIN))
            val greeting = service.callRestService()
            assertThat(greeting).isEqualTo("hello")
    }

}

```

Auto-configured Spring REST Docs Tests

You can use the `@AutoConfigureRestDocs` annotation to use [Spring REST Docs](#) in your tests with Mock MVC, REST Assured, or WebTestClient. It removes the need for the JUnit extension in Spring REST Docs.

`@AutoConfigureRestDocs` can be used to override the default output directory (`target/generated-snippets` if you are using Maven or `build/generated-snippets` if you are using Gradle). It can also be used to configure the host, scheme, and port that appears in any documented URIs.

Auto-configured Spring REST Docs Tests With Mock MVC

`@AutoConfigureRestDocs` customizes the `MockMvc` bean to use Spring REST Docs when testing servlet-based web applications. You can inject it by using `@Autowired` and use it in your tests as you normally would when using Mock MVC and Spring REST Docs, as shown in the following example:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(UserController.class)
@AutoConfigureRestDocs
class MyUserDocumentationTests {

    @Autowired
    private MockMvc mvc;

    @Test
    void listUsers() throws Exception {
        this.mvc.perform(get("/users").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk())
            .andDo(document("list-users"));
    }
}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest
import org.springframework.http.MediaType
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentation
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders
import org.springframework.test.web.servlet.result.MockMvcResultMatchers

@WebMvcTest(UserController::class)
@AutoConfigureRestDocs
class MyUserDocumentationTests(@Autowired val mvc: MockMvc) {

    @Test
    fun listUsers() {
        mvc.perform(MockMvcRequestBuilders.get("/users").accept(MediaType.TEXT_PLAIN))
            .andExpect(MockMvcResultMatchers.status().isOk)
            .andDo(MockMvcRestDocumentation.document("list-users"))
    }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, you can use a `RestDocsMockMvcConfigurationCustomizer` bean, as shown in the following example:

Java

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsMockMvcConfigurationCusto
mizer;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentationConfigurer;
import org.springframework.restdocs.templates.TemplateFormats;

@TestConfiguration(proxyBeanMethods = false)
public class MyRestDocsConfiguration implements RestDocsMockMvcConfigurationCustomizer
{

    @Override
    public void customize(MockMvcRestDocumentationConfigurer configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.restdocs.RestDocsMockMvcConfigurationCustomizer
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentationConfigurer
import org.springframework.restdocs.templates.TemplateFormats

@TestConfiguration(proxyBeanMethods = false)
class MyRestDocsConfiguration : RestDocsMockMvcConfigurationCustomizer {

    override fun customize(configurer: MockMvcRestDocumentationConfigurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown())
    }

}
```

If you want to make use of Spring REST Docs support for a parameterized output directory, you can create a `RestDocumentationResultHandler` bean. The auto-configuration calls `alwaysDo` with this result handler, thereby causing each `MockMvc` call to automatically generate the default snippets. The following example shows a `RestDocumentationResultHandler` being defined:

Java

```
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.restdocs.mockmvc.MockMvcRestDocumentation;
import org.springframework.restdocs.mockmvc.RestDocumentationResultHandler;

@TestConfiguration(proxyBeanMethods = false)
public class MyResultHandlerConfiguration {

    @Bean
    public RestDocumentationResultHandler restDocumentation() {
        return MockMvcRestDocumentation.document("{method-name}");
    }

}
```

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.context.annotation.Bean
import org.springframeworK.restdocs.mockmvc.MockMvcRestDocumentation
import org.springframeworK.restdocs.mockmvc.RestDocumentationResultHandler

@TestConfiguration(proxyBeanMethods = false)
class MyResultHandlerConfiguration {

    @Bean
    fun restDocumentation(): RestDocumentationResultHandler {
        return MockMvcRestDocumentation.document("{method-name}")
    }

}
```

Auto-configured Spring REST Docs Tests With WebTestClient

`@AutoConfigureRestDocs` can also be used with `WebTestClient` when testing reactive web applications. You can inject it by using `@Autowired` and use it in your tests as you normally would when using `@WebFluxTest` and Spring REST Docs, as shown in the following example:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.test.web.reactive.server.WebTestClient;

import static
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation.document;

@WebFluxTest
@AutoConfigureRestDocs
class MyUsersDocumentationTests {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    void listUsers() {
        this.webTestClient
            .get().uri("/")
            .exchange()
            .expectStatus()
                .isOk()
            .expectBody()
                .consumeWith(document("list-users"));
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest
import org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation
import org.springframework.test.web.reactive.server.WebTestClient

@WebFluxTest
@AutoConfigureRestDocs
class MyUsersDocumentationTests(@Autowired val webTestClient: WebTestClient) {

    @Test
    fun listUsers() {
        webTestClient
            .get().uri("/")
            .exchange()
            .expectStatus()
            .isOk
            .expectBody()
            .consumeWith(WebTestClientRestDocumentation.document("list-users"))
    }
}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, you can use a `RestDocsWebTestClientConfigurationCustomizer` bean, as shown in the following example:

Java

```
import
org.springframework.boot.test.autoconfigure.restdocs.RestDocsWebTestClientConfiguratio
nCustomizer;
import org.springframework.boot.test.context.TestConfiguration;
import
org.springframework.restdocs.webtestclient.WebTestClientRestDocumentationConfigurer;

@TestConfiguration(proxyBeanMethods = false)
public class MyRestDocsConfiguration implements
RestDocsWebTestClientConfigurationCustomizer {

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

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.restdocs.RestDocsWebClientConfigurationCustomizer
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.restdocs.webtestclient.WebTestClientRestDocumentationConfigurer

@TestConfiguration(proxyBeanMethods = false)
class MyRestDocsConfiguration : RestDocsWebClientConfigurationCustomizer {

    override fun customize(configurer: WebTestClientRestDocumentationConfigurer) {
        configurer.snippets().withEncoding("UTF-8")
    }

}
```

If you want to make use of Spring REST Docs support for a parameterized output directory, you can use a [WebTestClientBuilderCustomizer](#) to configure a consumer for every entity exchange result. The following example shows such a [WebTestClientBuilderCustomizer](#) being defined:

Java

```
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.test.web.reactive.server.WebTestClientBuilderCustomizer;
import org.springframework.context.annotation.Bean;

import static org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation.document;

@TestConfiguration(proxyBeanMethods = false)
public class MyWebTestClientBuilderCustomizerConfiguration {

    @Bean
    public WebTestClientBuilderCustomizer restDocumentation() {
        return (builder) -> builder.entityExchangeResultConsumer(document("{method-name}"));
    }

}
```

```
import org.springframework.boot.test.context.TestConfiguration
import
org.springframework.boot.test.web.reactive.server.WebTestClientBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation
import org.springframework.test.web.reactive.server.WebTestClient

@TestConfiguration(proxyBeanMethods = false)
class MyWebTestClientBuilderCustomizerConfiguration {

    @Bean
    fun restDocumentation(): WebTestClientBuilderCustomizer {
        return WebTestClientBuilderCustomizer { builder: WebTestClient.Builder ->
            builder.entityExchangeResultConsumer(
                WebTestClientRestDocumentation.document("{method-name}")
            )
        }
    }
}
```

Auto-configured Spring REST Docs Tests With REST Assured

`@AutoConfigureRestDocs` makes a `RequestSpecification` bean, preconfigured to use Spring REST Docs, available to your tests. You can inject it by using `@Autowired` and use it in your tests as you normally would when using REST Assured and Spring REST Docs, as shown in the following example:

Java

```
import io.restassured.specification.RequestSpecification;
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.server.LocalServerPort;

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

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

    @Test
    void listUsers(@Autowired RequestSpecification documentationSpec, @LocalServerPort
int port) {
        given(documentationSpec)
            .filter(document("list-users"))
        .when()
            .port(port)
            .get("/")
        .then().assertThat()
            .statusCode(is(200));
    }

}
```

```
import io.restassured.RestAssured
import io.restassured.specification.RequestSpecification
import org.hamcrest.Matchers
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.boot.test.web.server.LocalServerPort
import org.springframework.restdocs.restassured.RestAssuredRestDocumentation

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

    @Test
    fun listUsers(@Autowired documentationSpec: RequestSpecification?,
    @LocalServerPort port: Int) {
        RestAssured.given(documentationSpec)
            .filter(RestAssuredRestDocumentation.document("list-users"))
            .'when'()
            .port(port)["/"]
            .then().assertThat()
            .statusCode(Matchers.`is`(200))
    }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, a `RestDocsRestAssuredConfigurationCustomizer` bean can be used, as shown in the following example:

Java

```
import org.springframework.boot.test.autoconfigure.restdocs.RestDocsRestAssuredConfigurationCustomizer;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.restdocs.restassured.RestAssuredRestDocumentationConfigurer;
import org.springframework.restdocs.templates.TemplateFormats;

@TestConfiguration(proxyBeanMethods = false)
public class MyRestDocsConfiguration implements RestDocsRestAssuredConfigurationCustomizer {

    @Override
    public void customize(RestAssuredRestDocumentationConfigurer configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}
```

Kotlin

```
import org.springframework.boot.test.autoconfigure.restdocs.RestDocsRestAssuredConfigurationCustomizer
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.restdocs.restassured.RestAssuredRestDocumentationConfigurer
import org.springframework.restdocs.templates.TemplateFormats

@TestConfiguration(proxyBeanMethods = false)
class MyRestDocsConfiguration : RestDocsRestAssuredConfigurationCustomizer {

    override fun customize(configurer: RestAssuredRestDocumentationConfigurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown())
    }

}
```

Auto-configured Spring Web Services Tests

Auto-configured Spring Web Services Client Tests

You can use [@WebServiceClientTest](#) to test applications that call web services using the Spring Web Services project. By default, it configures a mock [WebServiceServer](#) bean and automatically customizes your [WebServiceTemplateBuilder](#). (For more about using Web Services with Spring Boot, see "[Web Services](#)".)

TIP

A list of the auto-configuration settings that are enabled by `@WebServiceClientTest` can be [found in the appendix](#).

The following example shows the `@WebServiceClientTest` annotation in use:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.webservices.client.WebServiceClientTest;
import org.springframework.ws.test.client.MockWebServiceServer;
import org.springframework.xml.transform.StringSource;

import static org.assertj.core.api.Assertions.assertThat;
import static org.springframework.ws.test.client.RequestMatchers.payload;
import static org.springframework.ws.test.client.ResponseCreators.withPayload;

@WebServiceClientTest(SomeWebService.class)
class MyWebServiceClientTests {

    @Autowired
    private MockWebServiceServer server;

    @Autowired
    private SomeWebService someWebService;

    @Test
    void mockServerCall() {
        this.server
            .expect(payload(new StringSource("<request/>")))
            .andRespond(withPayload(new
StringSource("<response><status>200</status></response>")));
        assertThat(this.someWebService.test())
            .extracting(Response::getStatus)
            .isEqualTo(200);
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.webservices.client.WebServiceClientTest
import org.springframework.ws.test.client.MockWebServiceServer
import org.springframework.ws.test.client.RequestMatchers
import org.springframework.ws.test.client.ResponseCreators
import org.springframework.xml.transform.StringSource

@WebServiceClientTest(SomeWebService::class)
class MyWebServiceClientTests(@Autowired val server: MockWebServiceServer, @Autowired
val someWebService: SomeWebService) {

    @Test
    fun mockServerCall() {
        server
            .expect(RequestMatchers.payload(StringSource("<request/>")))

        .andRespond(ResponseCreators.withPayload(StringSource("<response><status>200</status></response>")))

        assertThat(this.someWebService.test()).extracting(Response::status).isEqualTo(200)
    }
}
```

Auto-configured Spring Web Services Server Tests

You can use `@WebServiceServerTest` to test applications that implement web services using the Spring Web Services project. By default, it configures a `MockWebServiceClient` bean that can be used to call your web service endpoints. (For more about using Web Services with Spring Boot, see "[Web Services](#)".)

TIP

A list of the auto-configuration settings that are enabled by `@WebServiceServerTest` can be [found in the appendix](#).

The following example shows the `@WebServiceServerTest` annotation in use:

Java

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

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.webservices.server.WebServiceServerTest;
import org.springframework.ws.test.server.MockWebServiceClient;
import org.springframework.ws.test.server.RequestCreators;
import org.springframework.ws.test.server.ResponseMatchers;
import org.springframework.xml.transform.StringSource;

@WebServiceServerTest(ExampleEndpoint.class)
class MyWebServiceServerTests {

    @Autowired
    private MockWebServiceClient client;

    @Test
    void mockServerCall() {
        this.client
            .sendRequest(RequestCreators.withPayload(new
StringSource("<ExampleRequest/>")))
            .andExpect(ResponseMatchers.payload(new
StringSource("<ExampleResponse>42</ExampleResponse>")));
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.webservices.server.WebServiceServerTest
import org.springframework.ws.test.server.MockWebServiceClient
import org.springframework.ws.test.server.RequestCreators
import org.springframework.ws.test.server.ResponseMatchers
import org.springframework.xml.transform.StringSource

@WebServiceServerTest(ExampleEndpoint::class)
class MyWebServiceServerTests(@Autowired val client: MockWebServiceClient) {

    @Test
    fun mockServerCall() {
        client

        .sendRequest(RequestCreators.withPayload(StringSource("<ExampleRequest/>")))

        .andExpect(ResponseMatchers.payload(StringSource("<ExampleResponse>42</ExampleResponse
>")))
    }

}
```

Additional Auto-configuration and Slicing

Each slice provides one or more `@AutoConfigure…` annotations that namely defines the auto-configurations that should be included as part of a slice. Additional auto-configurations can be added on a test-by-test basis by creating a custom `@AutoConfigure…` annotation or by adding `@ImportAutoConfiguration` to the test as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.ImportAutoConfiguration;
import
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration;
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;

@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration.class)
class MyJdbcTests {

}
```

```
import org.springframework.boot.autoconfigure.ImportAutoConfiguration
import org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest

@JdbcTest
@ImportAutoConfiguration(IntegrationAutoConfiguration::class)
class MyJdbcTests
```

NOTE Make sure to not use the regular `@Import` annotation to import auto-configurations as they are handled in a specific way by Spring Boot.

Alternatively, additional auto-configurations can be added for any use of a slice annotation by registering them in a file stored in `META-INF/spring` as shown in the following example:

`META-INF/spring/org.springframework.boot.test.autoconfigure.jdbc.JdbcTest.imports`

```
com.example.IntegrationAutoConfiguration
```

In this example, the `com.example.IntegrationAutoConfiguration` is enabled on every test annotated with `@JdbcTest`.

TIP You can use comments with `#` in this file.

TIP A slice or `@AutoConfigure…` annotation can be customized this way as long as it is meta-annotated with `@ImportAutoConfiguration`.

User Configuration and Slicing

If you `structure your code` in a sensible way, your `@SpringBootApplication` class is `used by default` as the configuration of your tests.

It then becomes important not to litter the application's main class with configuration settings that are specific to a particular area of its functionality.

Assume that you are using Spring Data MongoDB, you rely on the auto-configuration for it, and you have enabled auditing. You could define your `@SpringBootApplication` as follows:

Java

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.mongodb.config.EnableMongoAuditing;

@SpringBootApplication
@EnableMongoAuditing
public class MyApplication {

    // ...

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.data.mongodb.config.EnableMongoAuditing

@SpringBootApplication
@EnableMongoAuditing
class MyApplication {

    // ...

}
```

Because this class is the source configuration for the test, any slice test actually tries to enable Mongo auditing, which is definitely not what you want to do. A recommended approach is to move that area-specific configuration to a separate `@Configuration` class at the same level as your application, as shown in the following example:

Java

```
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.config.EnableMongoAuditing;

@Configuration(proxyBeanMethods = false)
@EnableMongoAuditing
public class MyMongoConfiguration {

    // ...

}
```

```

import org.springframework.context.annotation.Configuration
import org.springframework.data.mongodb.config.EnableMongoAuditing

@Configuration(proxyBeanMethods = false)
@EnableMongoAuditing
class MyMongoConfiguration {

    // ...

}

```

NOTE Depending on the complexity of your application, you may either have a single `@Configuration` class for your customizations or one class per domain area. The latter approach lets you enable it in one of your tests, if necessary, with the `@Import` annotation. See [this how-to section](#) for more details on when you might want to enable specific `@Configuration` classes for slice tests.

Test slices exclude `@Configuration` classes from scanning. For example, for a `@WebMvcTest`, the following configuration will not include the given `WebMvcConfigurer` bean in the application context loaded by the test slice:

Java

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration(proxyBeanMethods = false)
public class MyWebConfiguration {

    @Bean
    public WebMvcConfigurer testConfigurer() {
        return new WebMvcConfigurer() {
            // ...
        };
    }

}

```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Configuration(proxyBeanMethods = false)
class MyWebConfiguration {

    @Bean
    fun testConfigurer(): WebMvcConfigurer {
        return object : WebMvcConfigurer {
            // ...
        }
    }

}
```

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

Java

```
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Component
public class MyWebMvcConfigurer implements WebMvcConfigurer {

    // ...

}
```

Kotlin

```
import org.springframework.stereotype.Component
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Component
class MyWebMvcConfigurer : WebMvcConfigurer {

    // ...

}
```

Another source of confusion is classpath scanning. Assume that, while you structured your code in a sensible way, you need to scan an additional package. Your application may resemble the following code:

Java

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

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

    // ...

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.context.annotation.ComponentScan

@SpringBootApplication
@ComponentScan("com.example.app", "com.example.another")
class MyApplication {

    // ...

}
```

Doing so effectively overrides the default component scan directive with the side effect of scanning those two packages regardless of the slice that you chose. For instance, a `@DataJpaTest` seems to suddenly scan components and user configurations of your application. Again, moving the custom directive to a separate class is a good way to fix this issue.

TIP If this is not an option for you, you can create a `@SpringBootConfiguration` somewhere in the hierarchy of your test so that it is used instead. Alternatively, you can specify a source for your test, which disables the behavior of finding a default one.

Using Spock to Test Spring Boot Applications

Spock 2.2 or later can be used to test a Spring Boot application. To do so, add a dependency on a `-groovy-4.0` version of Spock's `spock-spring` module to your application's build. `spock-spring` integrates Spring's test framework into Spock. See [the documentation for Spock's Spring module](#) for further details.

7.9.4. Testcontainers

The `Testcontainers` library provides a way to manage services running inside Docker containers. It integrates with JUnit, allowing you to write a test class that can start up a container before any of the tests run. Testcontainers is especially useful for writing integration tests that talk to a real backend service such as MySQL, MongoDB, Cassandra and others.

Testcontainers can be used in a Spring Boot test as follows:

Java

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import org.springframework.boot.test.context.SpringBootTest;

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>("neo4j:5");

    @Test
    void myTest() {
        // ...
    }

}
```

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.testcontainers.containers.Neo4jContainer
import org.testcontainers.junit.jupiter.Container
import org.testcontainers.junit.jupiter.Testcontainers

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Test
    fun myTest() {
        // ...
    }

    companion object {

        @Container
        val neo4j = Neo4jContainer("neo4j:5")

    }
}
```

This will start up a docker container running Neo4j (if Docker is running locally) before any of the tests are run. In most cases, you will need to configure the application to connect to the service running in the container.

Service Connections

A service connection is a connection to any remote service. Spring Boot's auto-configuration can consume the details of a service connection and use them to establish a connection to a remote service. When doing so, the connection details take precedence over any connection-related configuration properties.

When using Testcontainers, connection details can be automatically created for a service running in a container by annotating the container field in the test class.

Java

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Container
    @ServiceConnection
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>("neo4j:5");

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

Kotlin

```
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.testcontainers.service.connection.ServiceConnection
import org.testcontainers.containers.Neo4jContainer
import org.testcontainers.junit.jupiter.Container
import org.testcontainers.junit.jupiter.Testcontainers

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Test
    fun myTest() {
        // ...
    }

    companion object {

        @Container
        @ServiceConnection
        val neo4j = Neo4jContainer("neo4j:5")

    }
}
```

Thanks to `@ServiceConnection`, the above configuration allows Neo4j-related beans in the application to communicate with Neo4j running inside the Testcontainers-managed Docker container. This is done by automatically defining a `Neo4jConnectionDetails` bean which is then used by the Neo4j auto-configuration, overriding any connection-related configuration properties.

NOTE

You'll need to add the `spring-boot-testcontainers` module as a test dependency in order to use service connections with Testcontainers.

Service connection annotations are processed by `ContainerConnectionDetailsFactory` classes registered with `spring.factories`. A `ContainerConnectionDetailsFactory` can create a `ConnectionDetails` bean based on a specific `Container` subclass, or the Docker image name.

The following service connection factories are provided in the `spring-boot-testcontainers` jar:

Connection Details	Matched on
<code>ActiveMQConnectionDetails</code>	Containers named "symptoma/activemq"
<code>CassandraConnectionDetails</code>	Containers of type <code>CassandraContainer</code>
<code>CouchbaseConnectionDetails</code>	Containers of type <code>CouchbaseContainer</code>
<code>ElasticsearchConnectionDetails</code>	Containers of type <code>ElasticsearchContainer</code>

Connection Details	Matched on
<code>FlywayConnectionDetails</code>	Containers of type <code>JdbcDatabaseContainer</code>
<code>JdbcConnectionDetails</code>	Containers of type <code>JdbcDatabaseContainer</code>
<code>KafkaConnectionDetails</code>	Containers of type <code>org.testcontainers.containers.KafkaContainer</code> or <code>RedpandaContainer</code>
<code>LiquibaseConnectionDetails</code>	Containers of type <code>JdbcDatabaseContainer</code>
<code>MongoConnectionDetails</code>	Containers of type <code>MongoDBContainer</code>
<code>Neo4jConnectionDetails</code>	Containers of type <code>Neo4jContainer</code>
<code>OtlpMetricsConnectionDetails</code>	Containers named "otel/opentelemetry-collector-contrib"
<code>OtlpTracingConnectionDetails</code>	Containers named "otel/opentelemetry-collector-contrib"
<code>PulsarConnectionDetails</code>	Containers of type <code>PulsarContainer</code>
<code>R2dbcConnectionDetails</code>	Containers of type <code>MariaDBContainer</code> , <code>MSSQLServerContainer</code> , <code>MySQLContainer</code> , <code>OracleContainer</code> , or <code>PostgreSQLContainer</code>
<code>RabbitConnectionDetails</code>	Containers of type <code>RabbitMQContainer</code>
<code>RedisConnectionDetails</code>	Containers named "redis"
<code>ZipkinConnectionDetails</code>	Containers named "openzipkin/zipkin"

By default all applicable connection details beans will be created for a given `Container`. For example, a `PostgreSQLContainer` will create both `JdbcConnectionDetails` and `R2dbcConnectionDetails`.

TIP

If you want to create only a subset of the applicable types, you can use the `type` attribute of `@ServiceConnection`.

By default `Container.getDockerImageName().getRepository()` is used to obtain the name used to find connection details. The repository portion of the Docker image name ignores any registry and the version. This works as long as Spring Boot is able to get the instance of the `Container`, which is the case when using a `static` field like in the example above.

If you're using a `@Bean` method, Spring Boot won't call the bean method to get the Docker image name, because this would cause eager initialization issues. Instead, the return type of the bean method is used to find out which connection detail should be used. This works as long as you're using typed containers, e.g. `Neo4jContainer` or `RabbitMQContainer`. This stops working if you're using `GenericContainer`, e.g. with Redis, as shown in the following example:

Java

```
import org.testcontainers.containers.GenericContainer;

import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;
import org.springframework.context.annotation.Bean;

@TestConfiguration(proxyBeanMethods = false)
public class MyRedisConfiguration {

    @Bean
    @ServiceConnection(name = "redis")
    public GenericContainer<?> redisContainer() {
        return new GenericContainer<>("redis:7");
    }

}
```

Kotlin

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.service.connection.ServiceConnection
import org.springframework.context.annotation.Bean
import org.testcontainers.containers.GenericContainer

@TestConfiguration(proxyBeanMethods = false)
class MyRedisConfiguration {

    @Bean
    @ServiceConnection(name = "redis")
    fun redisContainer(): GenericContainer<*> {
        return GenericContainer("redis:7")
    }

}
```

Spring Boot can't tell from `GenericContainer` which container image is used, so the `name` attribute from `@ServiceConnection` must be used to provide that hint.

You can also can use the `name` attribute of `@ServiceConnection` to override which connection detail will be used, for example when using custom images. If you are using the Docker image `registry.mycompany.com/mirror/myredis`, you'd use `@ServiceConnection(name="redis")` to ensure `RedisConnectionDetails` are created.

Dynamic Properties

A slightly more verbose but also more flexible alternative to service connections is `@DynamicPropertySource`. A static `@DynamicPropertySource` method allows adding dynamic property

values to the Spring Environment.

Java

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Container
    static Neo4jContainer<?> neo4j = new Neo4jContainer<>("neo4j:5");

    @Test
    void myTest() {
        // ...
    }

    @DynamicPropertySource
    static void neo4jProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.neo4j.uri", neo4j::getBoltUrl);
    }
}
```

```

import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.context.DynamicPropertyRegistry
import org.springframework.test.context.DynamicPropertySource
import org.testcontainers.containers.Neo4jContainer
import org.testcontainers.junit.jupiter.Container
import org.testcontainers.junit.jupiter.Testcontainers

@Testcontainers
@SpringBootTest
class MyIntegrationTests {

    @Test
    fun myTest() {
        // ...
    }

    companion object {

        @Container
        val neo4j = Neo4jContainer("neo4j:5")

        @DynamicPropertySource
        fun neo4jProperties(registry: DynamicPropertyRegistry) {
            registry.add("spring.neo4j.uri") { neo4j.boltUrl }
        }
    }
}

```

The above configuration allows Neo4j-related beans in the application to communicate with Neo4j running inside the Testcontainers-managed Docker container.

7.9.5. Test Utilities

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

ConfigDataApplicationContextInitializer

[ConfigDataApplicationContextInitializer](#) is an [ApplicationContextInitializer](#) that you can apply to your tests to load Spring Boot [application.properties](#) files. You can use it when you do not need the full set of features provided by [@SpringBootTest](#), as shown in the following example:

Java

```
import org.springframework.boot.test.context.ConfigDataApplicationContextInitializer;
import org.springframework.test.context.ContextConfiguration;

@ContextConfiguration(classes = Config.class, initializers =
ConfigDataApplicationContextInitializer.class)
class MyConfigFileTests {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.context.ConfigDataApplicationContextInitializer
import org.springframework.test.context.ContextConfiguration

@ContextConfiguration(classes = [Config::class], initializers =
[ConfigDataApplicationContextInitializer::class])
class MyConfigFileTests {

    // ...

}
```

NOTE Using `ConfigDataApplicationContextInitializer` alone does not provide support for `@Value("${...}")` injection. Its only job is to ensure that `application.properties` files are loaded into Spring's `Environment`. For `@Value` support, you need to either additionally configure a `PropertySourcesPlaceholderConfigurer` or use `@SpringBootTest`, which auto-configures one for you.

TestPropertyValues

`TestPropertyValues` lets you quickly add properties to a `ConfigurableEnvironment` or `ConfigurableApplicationContext`. You can call it with `key=value` strings, as follows:

Java

```
import org.junit.jupiter.api.Test;  
  
import org.springframework.boot.test.util.TestPropertyValues;  
import org.springframework.mock.env.MockEnvironment;  
  
import static org.assertj.core.api.Assertions.assertThat;  
  
class MyEnvironmentTests {  
  
    @Test  
    void testPropertySources() {  
        MockEnvironment environment = new MockEnvironment();  
        TestPropertyValues.of("org=Spring", "name=Boot").applyTo(environment);  
        assertThat(environment.getProperty("name")).isEqualTo("Boot");  
    }  
  
}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat  
import org.junit.jupiter.api.Test  
import org.springframework.boot.test.util.TestPropertyValues  
import org.springframework.mock.env.MockEnvironment  
  
class MyEnvironmentTests {  
  
    @Test  
    fun testPropertySources() {  
        val environment = MockEnvironment()  
        TestPropertyValues.of("org=Spring", "name=Boot").applyTo(environment)  
        assertThat(environment.getProperty("name")).isEqualTo("Boot")  
    }  
  
}
```

OutputCapture

OutputCapture is a JUnit Extension that you can use to capture `System.out` and `System.err` output. To use it, add `@ExtendWith(OutputCaptureExtension.class)` and inject `CapturedOutput` as an argument to your test class constructor or test method as follows:

Java

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.boot.test.system.CapturedOutput;
import org.springframework.boot.test.system.OutputCaptureExtension;

import static org.assertj.core.api.Assertions.assertThat;

@ExtendWith(OutputCaptureExtension.class)
class MyOutputCaptureTests {

    @Test
    void testName(CapturedOutput output) {
        System.out.println("Hello World!");
        assertThat(output).contains("World");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.extension.ExtendWith
import org.springframework.boot.test.system.CapturedOutput
import org.springframework.boot.test.system.OutputCaptureExtension

@ExtendWith(OutputCaptureExtension::class)
class MyOutputCaptureTests {

    @Test
    fun testName(output: CapturedOutput?) {
        println("Hello World!")
        assertThat(output).contains("World")
    }

}
```

TestRestTemplate

TestRestTemplate is a convenience alternative to Spring's **RestTemplate** that is useful in integration tests. You can get a vanilla template or one that sends Basic HTTP authentication (with a username and password). In either case, the template is fault tolerant. This means that it behaves in a test-friendly way by not throwing exceptions on 4xx and 5xx errors. Instead, such errors can be detected through the returned **ResponseEntity** and its status code.

TIP Spring Framework 5.0 provides a new `WebTestClient` that works for `WebFlux` [integration tests](#) and both `WebFlux` and `MVC` end-to-end testing. It provides a fluent API for assertions, unlike `TestRestTemplate`.

It is recommended, but not mandatory, to use the Apache HTTP Client (version 5.1 or better). If you have that on your classpath, the `TestRestTemplate` responds by configuring the client appropriately. If you do use Apache's HTTP client, some additional test-friendly features are enabled:

- Redirects are not followed (so you can assert the response location).
- Cookies are ignored (so the template is stateless).

`TestRestTemplate` can be instantiated directly in your integration tests, as shown in the following example:

Java

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

import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.ResponseEntity;

import static org.assertj.core.api.Assertions.assertThat;

class MyTests {

    private final TestRestTemplate template = new TestRestTemplate();

    @Test
    void testRequest() {
        ResponseEntity<String> headers =
this.template.getForEntity("https://myhost.example.com/example", String.class);
        assertThat(headers.getHeaders().getLocation()).hasHost("other.example.com");
    }

}
```

Kotlin

```
import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.boot.test.web.client.TestRestTemplate

class MyTests {

    private val template = TestRestTemplate()

    @Test
    fun testRequest() {
        val headers = template.getForEntity("https://myhost.example.com/example",
String::class.java)
        assertThat(headers.headers.location).hasHost("other.example.com")
    }

}
```

Alternatively, if you use the `@SpringBootTest` annotation with `WebEnvironment.RANDOM_PORT` or `WebEnvironment.DEFINED_PORT`, you can inject a fully configured `TestRestTemplate` and start using it. If necessary, additional customizations can be applied through the `RestTemplateBuilder` bean. Any URLs that do not specify a host and port automatically connect to the embedded server, as shown in the following example:

Java

```
import java.time.Duration;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.http.HttpHeaders;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MySpringBootTests {

    @Autowired
    private TestRestTemplate template;

    @Test
    void testRequest() {
        HttpHeaders headers = this.template.getForEntity("/example",
String.class).getHeaders();
        assertThat(headers.getLocation()).hasHost("other.example.com");
    }

    @TestConfiguration(proxyBeanMethods = false)
    static class RestTemplateBuilderConfiguration {

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

```

import org.assertj.core.api.Assertions.assertThat
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.test.web.client.TestRestTemplate
import org.springframework.boot.web.client.RestTemplateBuilder
import org.springframework.context.annotation.Bean
import java.time.Duration

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MySpringBootTests(@Autowired val template: TestRestTemplate) {

    @Test
    fun testRequest() {
        val headers = template.getForEntity("/example", String::class.java).headers
        assertThat(headers.location).hasHost("other.example.com")
    }

    @TestConfiguration(proxyBeanMethods = false)
    internal class RestTemplateBuilderConfiguration {

        @Bean
        fun restTemplateBuilder(): RestTemplateBuilder {
            return RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1))
                .setReadTimeout(Duration.ofSeconds(1))
        }
    }
}

```

7.10. Docker Compose Support

Docker Compose is a popular technology that can be used to define and manage multiple containers for services that your application needs. A `compose.yml` file is typically created next to your application which defines and configures service containers.

A typical workflow with Docker Compose is to run `docker compose up`, work on your application with it connecting to started services, then run `docker compose down` when you are finished.

The `spring-boot-docker-compose` module can be included in a project to provide support for working with containers using Docker Compose. Add the module dependency to your build, as shown in the following listings for Maven and Gradle:

Maven

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

Gradle

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

When this module is included as a dependency Spring Boot will do the following:

- Search for a `compose.yml` and other common compose filenames in your working directory
- Call `docker compose up` with the discovered `compose.yml`
- Create service connection beans for each supported container
- Call `docker compose stop` when the application is shutdown

If the Docker Compose services are already running when starting the application, Spring Boot will only create the service connection beans for each supported container. It will not call `docker compose up` again and it will not call `docker compose stop` when the application is shutdown.

TIP

Repackaged archives do not contain Spring Boot's Docker Compose by default. If you want to use this support, you need to include it. When using the Maven plugin, set the `excludeDockerCompose` property to `false`. When using the Gradle plugin, [configure the task's classpath to include the `developmentOnly` configuration](#).

7.10.1. Prerequisites

You need to have the `docker` and `docker compose` (or `docker-compose`) CLI applications on your path. The minimum supported Docker Compose version is 2.2.0.

7.10.2. Service Connections

A service connection is a connection to any remote service. Spring Boot's auto-configuration can consume the details of a service connection and use them to establish a connection to a remote service. When doing so, the connection details take precedence over any connection-related configuration properties.

When using Spring Boot's Docker Compose support, service connections are established to the port mapped by the container.

NOTE

Docker compose is usually used in such a way that the ports inside the container are mapped to ephemeral ports on your computer. For example, a Postgres server may run inside the container using port 5432 but be mapped to a totally different port locally. The service connection will always discover and use the locally mapped port.

Service connections are established by using the image name of the container. The following service connections are currently supported:

Connection Details	Matched on
<code>ActiveMQConnectionDetails</code>	Containers named "symptoma/activemq"
<code>CassandraConnectionDetails</code>	Containers named "cassandra"
<code>ElasticsearchConnectionDetails</code>	Containers named "elasticsearch"
<code>JdbcConnectionDetails</code>	Containers named "gvenzl/oracle-free", "gvenzl/oracle-xe", "mariadb", "mssql/server", "mysql", or "postgres"
<code>MongoConnectionDetails</code>	Containers named "mongo"
<code>Neo4jConnectionDetails</code>	Containers named "neo4j"
<code>OtlpMetricsConnectionDetails</code>	Containers named "otel/opentelemetry-collector-contrib"
<code>OtlpTracingConnectionDetails</code>	Containers named "otel/opentelemetry-collector-contrib"
<code>PulsarConnectionDetails</code>	Containers named "apache-pulsar/pulsar"
<code>R2dbcConnectionDetails</code>	Containers named "gvenzl/oracle-free", "gvenzl/oracle-xe", "mariadb", "mssql/server", "mysql", or "postgres"
<code>RabbitConnectionDetails</code>	Containers named "rabbitmq"
<code>RedisConnectionDetails</code>	Containers named "redis"
<code>ZipkinConnectionDetails</code>	Containers named "openzipkin/zipkin".

7.10.3. Custom Images

Sometimes you may need to use your own version of an image to provide a service. You can use any custom image as long as it behaves in the same way as the standard image. Specifically, any environment variables that the standard image supports must also be used in your custom image.

If your image uses a different name, you can use a label in your `compose.yml` file so that Spring Boot can provide a service connection. Use a label named `org.springframework.boot.service-connection` to provide the service name.

For example:

```
services:  
  redis:  
    image: 'mycompany/mycustomredis:7.0'  
    ports:  
      - '6379'  
    labels:  
      org.springframework.boot.service-connection: redis
```

7.10.4. Skipping Specific Containers

If you have a container image defined in your `compose.yml` that you don't want connected to your application you can use a label to ignore it. Any container with labeled with `org.springframework.boot.ignore` will be ignored by Spring Boot.

For example:

```
services:  
  redis:  
    image: 'redis:7.0'  
    ports:  
      - '6379'  
    labels:  
      org.springframework.boot.ignore: true
```

7.10.5. Using a Specific Compose File

If your compose file is not in the same directory as your application, or if it's named differently, you can use `spring.docker.compose.file` in your `application.properties` or `application.yaml` to point to a different file. Properties can be defined as an exact path or a path that's relative to your application.

For example:

Properties

```
spring.docker.compose.file=../my-compose.yml
```

Yaml

```
spring:  
  docker:  
    compose:  
      file: "../my-compose.yml"
```

7.10.6. Waiting for Container Readiness

Containers started by Docker Compose may take some time to become fully ready. The recommended way of checking for readiness is to add a `healthcheck` section under the service definition in your `compose.yml` file.

Since it's not uncommon for `healthcheck` configuration to be omitted from `compose.yml` files, Spring Boot also checks directly for service readiness. By default, a container is considered ready when a TCP/IP connection can be established to its mapped port.

You can disable this on a per-container basis by adding a `org.springframework.boot.readiness-check.tcp.disable` label in your `compose.yml` file.

For example:

```
services:  
  redis:  
    image: 'redis:7.0'  
    ports:  
      - '6379'  
    labels:  
      org.springframework.boot.readiness-check.tcp.disable: true
```

You can also change timeout values in your `application.properties` or `application.yaml` file:

Properties

```
spring.docker.compose.readiness.tcp.connect-timeout=10s  
spring.docker.compose.readiness.tcp.read-timeout=5s
```

Yaml

```
spring:  
  docker:  
    compose:  
      readiness:  
        tcp:  
          connect-timeout: 10s  
          read-timeout: 5s
```

The overall timeout can be configured using `spring.docker.compose.readiness.timeout`.

7.10.7. Controlling the Docker Compose Lifecycle

By default Spring Boot calls `docker compose up` when your application starts and `docker compose stop` when it's shut down. If you prefer to have different lifecycle management you can use the `spring.docker.compose.lifecycle-management` property.

The following values are supported:

- **none** - Do not start or stop Docker Compose
- **start-only** - Start Docker Compose when the application starts and leave it running
- **start-and-stop** - Start Docker Compose when the application starts and stop it when the JVM exits

In addition you can use the `spring.docker.compose.start.command` property to change whether `docker compose up` or `docker compose start` is used. The `spring.docker.compose.stop.command` allows you to configure if `docker compose down` or `docker compose stop` is used.

The following example shows how lifecycle management can be configured:

Properties

```
spring.docker.compose.lifecycle-management=start-and-stop
spring.docker.compose.start.command=start
spring.docker.compose.stop.command=down
spring.docker.compose.stop.timeout=1m
```

Yaml

```
spring:
  docker:
    compose:
      lifecycle-management: start-and-stop
      start:
        command: start
      stop:
        command: down
        timeout: 1m
```

7.10.8. Activating Docker Compose Profiles

Docker Compose profiles are similar to Spring profiles in that they let you adjust your Docker Compose configuration for specific environments. If you want to activate a specific Docker Compose profile you can use the `spring.docker.compose.profiles.active` property in your `application.properties` or `application.yaml` file:

Properties

```
spring.docker.compose.profiles.active=myprofile
```

Yaml

```
spring:  
  docker:  
    compose:  
      profiles:  
        active: "myprofile"
```

7.10.9. Using Docker Compose in Tests

By default, Spring Boot’s Docker Compose support is disabled when running tests.

To enable Docker Compose support in tests, set `spring.docker.compose.skip.in-tests` to `false`.

When using Gradle, you also need to change the configuration of the `spring-boot-docker-compose` dependency from `developmentOnly` to `testAndDevelopmentOnly`:

Gradle

```
dependencies {  
  testAndDevelopmentOnly("org.springframework.boot:spring-boot-docker-compose")  
}
```

7.11. Testcontainers Support

As well as [using Testcontainers for integration testing](#), it’s also possible to use them at development time. The next sections will provide more details about that.

7.11.1. Using Testcontainers at Development Time

This approach allows developers to quickly start containers for the services that the application depends on, removing the need to manually provision things like database servers. Using Testcontainers in this way provides functionality similar to Docker Compose, except that your container configuration is in Java rather than YAML.

To use Testcontainers at development time you need to launch your application using your “test” classpath rather than “main”. This will allow you to access all declared test dependencies and give you a natural place to write your test configuration.

To create a test launchable version of your application you should create an “Application” class in the `src/test` directory. For example, if your main application is in `src/main/java/com/example/MyApplication.java`, you should create `src/test/java/com/example/TestMyApplication.java`

The `TestMyApplication` class can use the `SpringApplication.from(...)` method to launch the real application:

Java

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

public class TestMyApplication {

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

}
```

Kotlin

```
import org.springframework.boot.fromApplication

fun main(args: Array<String>) {
    fromApplication<MyApplication>().run(*args)
}
```

You'll also need to define the `Container` instances that you want to start along with your application. To do this, you need to make sure that the `spring-boot-testcontainers` module has been added as a `test` dependency. Once that has been done, you can create a `@TestConfiguration` class that declares `@Bean` methods for the containers you want to start.

You can also annotate your `@Bean` methods with `@ServiceConnection` in order to create `ConnectionDetails` beans. See the `service connections` section for details of the supported technologies.

A typical Testcontainers configuration would look like this:

Java

```
import org.testcontainers.containers.Neo4jContainer;

import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;
import org.springframework.context.annotation.Bean;

@TestConfiguration(proxyBeanMethods = false)
public class MyContainersConfiguration {

    @Bean
    @ServiceConnection
    public Neo4jContainer<?> neo4jContainer() {
        return new Neo4jContainer<>("neo4j:5");
    }

}
```

```
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.service.connection.ServiceConnection
import org.springframework.context.annotation.Bean
import org.testcontainers.containers.Neo4jContainer

@TestConfiguration(proxyBeanMethods = false)
class MyContainersConfiguration {

    @Bean
    @ServiceConnection
    fun neo4jContainer(): Neo4jContainer<*> {
        return Neo4jContainer("neo4j:5")
    }

}
```

NOTE The lifecycle of `Container` beans is automatically managed by Spring Boot. Containers will be started and stopped automatically.

TIP You can use the `spring.testcontainers.beans.startup` property to change how containers are started. By default `sequential` startup is used, but you may also choose `parallel` if you wish to start multiple containers in parallel.

Once you have defined your test configuration, you can use the `with(...)` method to attach it to your test launcher:

Java

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

public class TestMyApplication {

    public static void main(String[] args) {

        SpringApplication.from(MyApplication::main).with(MyContainersConfiguration.class).run(
            args);
    }

}
```

Kotlin

```
import org.springframework.boot.fromApplication
import org.springframework.boot.with

fun main(args: Array<String>) {
    fromApplication<MyApplication>().with(MyContainersConfiguration::class).run(*args)
}
```

You can now launch `TestMyApplication` as you would any regular Java `main` method application to start your application and the containers that it needs to run.

TIP You can use the Maven goal `spring-boot:test-run` or the Gradle task `bootTestRun` to do this from the command line.

Contributing Dynamic Properties at Development Time

If you want to contribute dynamic properties at development time from your `Container @Bean` methods, you can do so by injecting a `DynamicPropertyRegistry`. This works in a similar way to the `@DynamicPropertySource annotation` that you can use in your tests. It allows you to add properties that will become available once your container has started.

A typical configuration would look like this:

Java

```
import org.testcontainers.containers.MongoDBContainer;

import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.test.context.DynamicPropertyRegistry;

@TestConfiguration(proxyBeanMethods = false)
public class MyContainersConfiguration {

    @Bean
    public MongoDBContainer mongoDbContainer(DynamicPropertyRegistry properties) {
        MongoDBContainer container = new MongoDBContainer("mongo:5.0");
        properties.add("spring.data.mongodb.host", container::getHost);
        properties.add("spring.data.mongodb.port", container::getFirstMappedPort);
        return container;
    }

}
```

```

import org.springframework.boot.test.context.TestConfiguration
import org.springframework.context.annotation.Bean
import org.springframework.test.context.DynamicPropertyRegistry
import org.testcontainers.containers.MongoDBContainer

@TestConfiguration(proxyBeanMethods = false)
class MyContainersConfiguration {

    @Bean
    fun monogDbContainer(properties: DynamicPropertyRegistry): MongoDBContainer {
        var container = MongoDBContainer("mongo:5.0")
        properties.add("spring.data.mongodb.host", container::getHost)
        properties.add("spring.data.mongodb.port", container::getFirstMappedPort)
        return container
    }

}

```

NOTE Using a `@ServiceConnection` is recommended whenever possible, however, dynamic properties can be a useful fallback for technologies that don't yet have `@ServiceConnection` support.

Importing Testcontainer Declaration Classes

A common pattern when using Testcontainers is to declare `Container` instances as static fields. Often these fields are defined directly on the test class. They can also be declared on a parent class or on an interface that the test implements.

For example, the following `MyContainers` interface declares `mongo` and `neo4j` containers:

```

import org.testcontainers.containers.MongoDBContainer;
import org.testcontainers.containers.Neo4jContainer;
import org.testcontainers.junit.jupiter.Container;

import org.springframework.boot.testcontainers.service.connection.ServiceConnection;

public interface MyContainers {

    @Container
    @ServiceConnection
    MongoDBContainer mongoContainer = new MongoDBContainer("mongo:5.0");

    @Container
    @ServiceConnection
    Neo4jContainer<?> neo4jContainer = new Neo4jContainer<>("neo4j:5");

}

```

If you already have containers defined in this way, or you just prefer this style, you can import these declaration classes rather than defining your containers as `@Bean` methods. To do so, add the `@ImportTestcontainers` annotation to your test configuration class:

Java

```

import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.context.ImportTestcontainers;

@TestConfiguration(proxyBeanMethods = false)
@ImportTestcontainers(MyContainers.class)
public class MyContainersConfiguration {

}

```

Kotlin

```

import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.context.ImportTestcontainers

@TestConfiguration(proxyBeanMethods = false)
@ImportTestcontainers(MyContainers::class)
class MyContainersConfiguration

```

TIP If you don't intend to use the [service connections feature](#) but want to use `@DynamicPropertySource` instead, remove the `@ServiceConnection` annotation from the `Container` fields. You can also add `@DynamicPropertySource` annotated methods to your declaration class.

Using DevTools with Testcontainers at Development Time

When using devtools, you can annotate beans and bean methods with `@RestartScope`. Such beans won't be recreated when the devtools restart the application. This is especially useful for Testcontainer `Container` beans, as they keep their state despite the application restart.

Java

```
import org.testcontainers.containers.MongoDBContainer;

import org.springframework.boot.devtools.restart.RestartScope;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;
import org.springframework.context.annotation.Bean;

@TestConfiguration(proxyBeanMethods = false)
public class MyContainersConfiguration {

    @Bean
    @RestartScope
    @ServiceConnection
    public MongoDBContainer mongoDbContainer() {
        return new MongoDBContainer("mongo:5.0");
    }

}
```

Kotlin

```
import org.springframework.boot.devtools.restart.RestartScope
import org.springframework.boot.test.context.TestConfiguration
import org.springframework.boot.testcontainers.service.connection.ServiceConnection
import org.springframework.context.annotation.Bean
import org.testcontainers.containers.MongoDBContainer

@TestConfiguration(proxyBeanMethods = false)
class MyContainersConfiguration {

    @Bean
    @RestartScope
    @ServiceConnection
    fun monogDbContainer(): MongoDBContainer {
        return MongoDBContainer("mongo:5.0")
    }

}
```

WARNING

If you’re using Gradle and want to use this feature, you need to change the configuration of the `spring-boot-devtools` dependency from `developmentOnly` to `testAndDevelopmentOnly`. With the default scope of `developmentOnly`, the `bootTestRun` task will not pick up changes in your code, as the devtools are not active.

7.12. Creating Your Own Auto-configuration

If you work in a company that develops shared libraries, or if you work on an open-source or commercial library, you might want to develop your own auto-configuration. Auto-configuration classes can be bundled in external jars and still be picked up by Spring Boot.

Auto-configuration can be associated to a “starter” that provides the auto-configuration code as well as the typical libraries that you would use with it. We first cover what you need to know to build your own auto-configuration and then we move on to the [typical steps required to create a custom starter](#).

7.12.1. Understanding Auto-configured Beans

Classes that implement auto-configuration are annotated with `@AutoConfiguration`. This annotation itself is meta-annotated with `@Configuration`, making auto-configuration standard `@Configuration` classes. Additional `@Conditional` annotations are used to constrain when the auto-configuration should apply. Usually, auto-configuration classes use `@ConditionalOnClass` and `@ConditionalOnMissingBean` annotations. This ensures that auto-configuration applies only when relevant classes are found and when you have not declared your own `@Configuration`.

You can browse the source code of `spring-boot-autoconfigure` to see the `@AutoConfiguration` classes that Spring provides (see the `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file).

7.12.2. Locating Auto-configuration Candidates

Spring Boot checks for the presence of a `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file within your published jar. The file should list your configuration classes, with one class name per line, as shown in the following example:

```
com.mycorp.libx.autoconfigure.LibXAutoConfiguration  
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

TIP

You can add comments to the imports file using the `#` character.

NOTE

Auto-configurations must be loaded *only* by being named in the imports file. Make sure that they are defined in a specific package space and that they are never the target of component scanning. Furthermore, auto-configuration classes should not enable component scanning to find additional components. Specific `@Import` annotations should be used instead.

If your configuration needs to be applied in a specific order, you can use the `before`, `beforeName`, `after` and `afterName` attributes on the `@AutoConfiguration` annotation or the dedicated `@AutoConfigureBefore` and `@AutoConfigureAfter` annotations. For example, if you provide web-specific configuration, your class may need to be applied after `WebMvcAutoConfiguration`.

If you want to order certain auto-configurations that should not have any direct knowledge of each other, you can also use `@AutoConfigureOrder`. That annotation has the same semantic as the regular `@Order` annotation but provides a dedicated order for auto-configuration classes.

As with standard `@Configuration` classes, the order in which auto-configuration classes are applied only affects the order in which their beans are defined. The order in which those beans are subsequently created is unaffected and is determined by each bean's dependencies and any `@DependsOn` relationships.

7.12.3. Condition Annotations

You almost always want to include one or more `@Conditional` annotations on your auto-configuration class. The `@ConditionalOnMissingBean` annotation is one common example that is used to allow developers to override auto-configuration if they are not happy with your defaults.

Spring Boot includes a number of `@Conditional` annotations that you can reuse in your own code by annotating `@Configuration` classes or individual `@Bean` methods. These annotations include:

- [Class Conditions](#)
- [Bean Conditions](#)
- [Property Conditions](#)
- [Resource Conditions](#)
- [Web Application Conditions](#)
- [SpEL Expression Conditions](#)

Class Conditions

The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations let `@Configuration` classes be included based on the presence or absence of specific classes. Due to the fact that annotation metadata is parsed by using `ASM`, you can use the `value` attribute to refer to the real class, even though that class might not actually appear on the running application classpath. You can also use the `name` attribute if you prefer to specify the class name by using a `String` value.

This mechanism does not apply the same way to `@Bean` methods where typically the return type is the target of the condition: before the condition on the method applies, the JVM will have loaded the class and potentially processed method references which will fail if the class is not present.

To handle this scenario, a separate `@Configuration` class can be used to isolate the condition, as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.AutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
// Some conditions ...
public class MyAutoConfiguration {

    // Auto-configured beans ...

    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(SomeService.class)
    public static class SomeServiceConfiguration {

        @Bean
        @ConditionalOnMissingBean
        public SomeService someService() {
            return new SomeService();
        }
    }

}
```

```

import org.springframework.boot.autoconfigure.condition.ConditionalOnClass
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

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

    // Auto-configured beans ...
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass(SomeService::class)
    class SomeServiceConfiguration {

        @Bean
        @ConditionalOnMissingBean
        fun someService(): SomeService {
            return SomeService()
        }

    }

}

```

TIP If you use `@ConditionalOnClass` or `@ConditionalOnMissingClass` as a part of a meta-annotation to compose your own composed annotations, you must use `name` as referring to the class in such a case is not handled.

Bean Conditions

The `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations let a bean be included based on the presence or absence of specific beans. You can use the `value` attribute to specify beans by type or `name` to specify beans by name. The `search` attribute lets you limit the `ApplicationContext` hierarchy that should be considered when searching for beans.

When placed on a `@Bean` method, the target type defaults to the return type of the method, as shown in the following example:

Java

```
import org.springframework.boot.autoconfigure.AutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;

@AutoConfiguration
public class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public SomeService someService() {
        return new SomeService();
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    fun someService(): SomeService {
        return SomeService()
    }

}
```

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

TIP You need to be very careful about the order in which bean definitions are added, as these conditions are evaluated based on what has been processed so far. For this reason, we recommend using only `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations on auto-configuration classes (since these are guaranteed to load after any user-defined bean definitions have been added).

NOTE `@ConditionalOnBean` and `@ConditionalOnMissingBean` do not prevent `@Configuration` classes from being created. The only difference between using these conditions at the class level and marking each contained `@Bean` method with the annotation is that the former prevents registration of the `@Configuration` class as a bean if the condition does not match.

TIP

When declaring a `@Bean` method, provide as much type information as possible in the method's return type. For example, if your bean's concrete class implements an interface the bean method's return type should be the concrete class and not the interface. Providing as much type information as possible in `@Bean` methods is particularly important when using bean conditions as their evaluation can only rely upon to type information that is available in the method signature.

Property Conditions

The `@ConditionalOnProperty` annotation lets configuration be included based on a Spring Environment property. Use the `prefix` and `name` attributes to specify the property that should be checked. By default, any property that exists and is not equal to `false` is matched. You can also create more advanced checks by using the `havingValue` and `matchIfMissing` attributes.

If multiple names are given in the `name` attribute, all of the properties have to pass the test for the condition to match.

Resource Conditions

The `@ConditionalOnResource` annotation lets configuration be included only when a specific resource is present. Resources can be specified by using the usual Spring conventions, as shown in the following example: `file:/home/user/test.dat`.

Web Application Conditions

The `@ConditionalOnWebApplication` and `@ConditionalOnNotWebApplication` annotations let configuration be included depending on whether the application is a web application. A servlet-based web application is any application that uses a Spring `WebApplicationContext`, defines a `session` scope, or has a `ConfigurableWebEnvironment`. A reactive web application is any application that uses a `ReactiveWebApplicationContext`, or has a `ConfigurableReactiveWebEnvironment`.

The `@ConditionalOnWarDeployment` and `@ConditionalOnNotWarDeployment` annotations let configuration be included depending on whether the application is a traditional WAR application that is deployed to a servlet container. This condition will not match for applications that are run with an embedded web server.

SpEL Expression Conditions

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

NOTE

Referencing a bean in the expression will cause that bean to be initialized very early in context refresh processing. As a result, the bean won't be eligible for post-processing (such as configuration properties binding) and its state may be incomplete.

7.12.4. Testing your Auto-configuration

An auto-configuration can be affected by many factors: user configuration (`@Bean` definition and

[Environment](#) customization), condition evaluation (presence of a particular library), and others. Concretely, each test should create a well defined [ApplicationContext](#) that represents a combination of those customizations. [ApplicationContextRunner](#) provides a great way to achieve that.

WARNING

[ApplicationContextRunner](#) doesn't work when running the tests in a native image.

[ApplicationContextRunner](#) is usually defined as a field of the test class to gather the base, common configuration. The following example makes sure that [MyServiceAutoConfiguration](#) is always invoked:

Java

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

Kotlin

```
val contextRunner = ApplicationContextRunner()
    .withConfiguration(AutoConfigurations.of(MyServiceAutoConfiguration::class.java))
```

TIP

If multiple auto-configurations have to be defined, there is no need to order their declarations as they are invoked in the exact same order as when running the application.

Each test can use the runner to represent a particular use case. For instance, the sample below invokes a user configuration ([UserConfiguration](#)) and checks that the auto-configuration backs off properly. Invoking [run](#) provides a callback context that can be used with [AssertJ](#).

Java

```
@Test
void defaultServiceBacksOff() {
    this.contextRunner.withUserConfiguration(UserConfiguration.class).run((context) ->
{
    assertThat(context).hasSingleBean(MyService.class);

    assertThat(context.getBean("myCustomService")).isSameAs(context.getBean(MyService.class));
    });
}

@Configuration(proxyBeanMethods = false)
static class UserConfiguration {

    @Bean
    MyService myCustomService() {
        return new MyService("mine");
    }

}
```

Kotlin

```
@Test
fun defaultServiceBacksOff() {
    contextRunner.withUserConfiguration(UserConfiguration::class.java)
        .run { context: AssertableApplicationContext ->
            assertThat(context).hasSingleBean(MyService::class.java)
            assertThat(context.getBean("myCustomService"))
                .isSameAs(context.getBean(MyService::class.java))
        }
}

@Configuration(proxyBeanMethods = false)
internal class UserConfiguration {

    @Bean
    fun myCustomService(): MyService {
        return MyService("mine")
    }

}
```

It is also possible to easily customize the [Environment](#), as shown in the following example:

Java

```
@Test
void serviceNameCanBeConfigured() {
    this.contextRunner.withPropertyValues("user.name=test123").run((context) -> {
        assertThat(context).hasSingleBean(MyService.class);
        assertThat(context.getBean(MyService.class).getName()).isEqualTo("test123");
    });
}
```

Kotlin

```
@Test
fun serviceNameCanBeConfigured() {
    contextRunner.withPropertyValues("user.name=test123").run { context:
    AssertableApplicationContext ->
        assertThat(context).hasSingleBean(MyService::class.java)
        assertThat(context.getBean(MyService::class.java).name).isEqualTo("test123")
    }
}
```

The runner can also be used to display the [ConditionEvaluationReport](#). The report can be printed at `INFO` or `DEBUG` level. The following example shows how to use the [ConditionEvaluationReportLoggingListener](#) to print the report in auto-configuration tests.

Java

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

import
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListene
r;
import org.springframework.boot.logging.LogLevel;
import org.springframework.boot.test.context.runner.ApplicationContextRunner;

class MyConditionEvaluationReportingTests {

    @Test
    void autoConfigTest() {
        new ApplicationContextRunner()

        .withInitializer(ConditionEvaluationReportLoggingListener.forLogLevel(LogLevel.INFO))
        .run((context) -> {
            // Test something...
        });
    }

}
```

```
import org.junit.jupiter.api.Test
import
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListene
r
import org.springframework.boot.logging.LogLevel
import org.springframework.boot.test.context.assertj.AssertableApplicationContext
import org.springframework.boot.test.context.runner.ApplicationContextRunner

class MyConditionEvaluationReportingTests {

    @Test
    fun autoConfigTest() {
        ApplicationContextRunner()
            .withInitializer(ConditionEvaluationReportLoggingListener.forLogLevel(LogLevel.INFO))
            .run { context: AssertableApplicationContext? -> }
    }

}
```

Simulating a Web Context

If you need to test an auto-configuration that only operates in a servlet or reactive web application context, use the [WebApplicationContextRunner](#) or [ReactiveWebApplicationContextRunner](#) respectively.

Overriding the Classpath

It is also possible to test what happens when a particular class and/or package is not present at runtime. Spring Boot ships with a [FilteredClassLoader](#) that can easily be used by the runner. In the following example, we assert that if `MyService` is not present, the auto-configuration is properly disabled:

Java

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

```

@Test
fun serviceIsIgnoredIfLibraryIsNotPresent() {
    contextRunner.withClassLoader(FilteredClassLoader(MyService::class.java))
        .run { context: AssertableApplicationContext? ->
            assertThat(context).doesNotHaveBean("myService")
        }
}

```

7.12.5. Creating Your Own Starter

A typical Spring Boot starter contains code to auto-configure and customize the infrastructure of a given technology, let's call that "acme". To make it easily extensible, a number of configuration keys in a dedicated namespace can be exposed to the environment. Finally, a single "starter" dependency is provided to help users get started as easily as possible.

Concretely, a custom starter can contain the following:

- The `autoconfigure` module that contains the auto-configuration code for "acme".
- The `starter` module that provides a dependency to the `autoconfigure` module as well as "acme" and any additional dependencies that are typically useful. In a nutshell, adding the starter should provide everything needed to start using that library.

This separation in two modules is in no way necessary. If "acme" has several flavors, options or optional features, then it is better to separate the auto-configuration as you can clearly express the fact some features are optional. Besides, you have the ability to craft a starter that provides an opinion about those optional dependencies. At the same time, others can rely only on the `autoconfigure` module and craft their own starter with different opinions.

If the auto-configuration is relatively straightforward and does not have optional features, merging the two modules in the starter is definitely an option.

Naming

You should make sure to provide a proper namespace for your starter. Do not start your module names with `spring-boot`, even if you use a different Maven `groupId`. We may offer official support for the thing you auto-configure in the future.

As a rule of thumb, you should name a combined module after the starter. For example, assume that you are creating a starter for "acme" and that you name the auto-configure module `acme-spring-boot` and the starter `acme-spring-boot-starter`. If you only have one module that combines the two, name it `acme-spring-boot-starter`.

Configuration keys

If your starter provides configuration keys, use a unique namespace for them. In particular, do not include your keys in the namespaces that Spring Boot uses (such as `server`, `management`, `spring`, and so on). If you use the same namespace, we may modify these namespaces in the future in ways that

break your modules. As a rule of thumb, prefix all your keys with a namespace that you own (for example `acme`).

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

Java

```
import java.time.Duration;

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

@ConfigurationProperties("acme")
public class AcmeProperties {

    /**
     * Whether to check the location of acme resources.
     */
    private boolean checkLocation = true;

    /**
     * Timeout for establishing a connection to the acme server.
     */
    private Duration loginTimeout = Duration.ofSeconds(3);

    public boolean isCheckLocation() {
        return this.checkLocation;
    }

    public void setCheckLocation(boolean checkLocation) {
        this.checkLocation = checkLocation;
    }

    public Duration getLoginTimeout() {
        return this.loginTimeout;
    }

    public void setLoginTimeout(Duration loginTimeout) {
        this.loginTimeout = loginTimeout;
    }

}
```

```

import org.springframework.boot.context.properties.ConfigurationProperties
import java.time.Duration

@ConfigurationProperties("acme")
class AcmeProperties {

    /**
     * Whether to check the location of acme resources.
     */
    var isCheckLocation: Boolean = true,

    /**
     * Timeout for establishing a connection to the acme server.
     */
    var loginTimeout: Duration = Duration.ofSeconds(3))

```

NOTE

You should only use plain text with `@ConfigurationProperties` field Javadoc, since they are not processed before being added to the JSON.

Here are some rules we follow internally to make sure descriptions are consistent:

- Do not start the description by "The" or "A".
- For `boolean` types, start the description with "Whether" or "Enable".
- For collection-based types, start the description with "Comma-separated list"
- Use `java.time.Duration` rather than `long` and describe the default unit if it differs from milliseconds, such as "If a duration suffix is not specified, seconds will be used".
- Do not provide the default value in the description unless it has to be determined at runtime.

Make sure to [trigger meta-data generation](#) so that IDE assistance is available for your keys as well. You may want to review the generated metadata (`META-INF/spring-configuration-metadata.json`) to make sure your keys are properly documented. Using your own starter in a compatible IDE is also a good idea to validate that quality of the metadata.

The “autoconfigure” Module

The `autoconfigure` module contains everything that is necessary to get started with the library. It may also contain configuration key definitions (such as `@ConfigurationProperties`) and any callback interface that can be used to further customize how the components are initialized.

TIP

You should mark the dependencies to the library as optional so that you can include the `autoconfigure` module in your projects more easily. If you do it that way, the library is not provided and, by default, Spring Boot backs off.

Spring Boot uses an annotation processor to collect the conditions on auto-configurations in a metadata file (`META-INF/spring-autoconfigure-metadata.properties`). If that file is present, it is used

to eagerly filter auto-configurations that do not match, which will improve startup time.

When building with Maven, it is recommended to add the following dependency in a module that contains auto-configurations:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure-processor</artifactId>
    <optional>true</optional>
</dependency>
```

If you have defined auto-configurations directly in your application, make sure to configure the [spring-boot-maven-plugin](#) to prevent the [repackage](#) goal from adding the dependency into the uber jar:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.springframework.boot</groupId>
                            <artifactId>spring-boot-autoconfigure-
processor</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

With Gradle, the dependency should be declared in the [annotationProcessor](#) configuration, as shown in the following example:

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

Starter Module

The starter is really an empty jar. Its only purpose is to provide the necessary dependencies to work with the library. You can think of it as an opinionated view of what is required to get started.

Do not make assumptions about the project in which your starter is added. If the library you are auto-configuring typically requires other starters, mention them as well. Providing a proper set of *default* dependencies may be hard if the number of optional dependencies is high, as you should avoid including dependencies that are unnecessary for a typical usage of the library. In other words, you should not include optional dependencies.

NOTE

Either way, your starter must reference the core Spring Boot starter ([spring-boot-starter](#)) directly or indirectly (there is no need to add it if your starter relies on another starter). If a project is created with only your custom starter, Spring Boot's core features will be honoured by the presence of the core starter.

7.13. Kotlin Support

[Kotlin](#) is a statically-typed language targeting the JVM (and other platforms) which allows writing concise and elegant code while providing [interoperability](#) with existing libraries written in Java.

Spring Boot provides Kotlin support by leveraging the support in other Spring projects such as Spring Framework, Spring Data, and Reactor. See the [Spring Framework Kotlin support documentation](#) for more information.

The easiest way to start with Spring Boot and Kotlin is to follow [this comprehensive tutorial](#). You can create new Kotlin projects by using [start.spring.io](#). Feel free to join the #spring channel of [Kotlin Slack](#) or ask a question with the [spring](#) and [kotlin](#) tags on [Stack Overflow](#) if you need support.

7.13.1. Requirements

Spring Boot requires at least Kotlin 1.7.x and manages a suitable Kotlin version through dependency management. To use Kotlin, [org.jetbrains.kotlin:kotlin-stdlib](#) and [org.jetbrains.kotlin:kotlin-reflect](#) must be present on the classpath. The [kotlin-stdlib](#) variants [kotlin-stdlib-jdk7](#) and [kotlin-stdlib-jdk8](#) can also be used.

Since [Kotlin classes are final by default](#), you are likely to want to configure [kotlin-spring](#) plugin in order to automatically open Spring-annotated classes so that they can be proxied.

[Jackson's Kotlin module](#) is required for serializing / deserializing JSON data in Kotlin. It is automatically registered when found on the classpath. A warning message is logged if Jackson and Kotlin are present but the Jackson Kotlin module is not.

TIP

These dependencies and plugins are provided by default if one bootstraps a Kotlin project on [start.spring.io](#).

7.13.2. Null-safety

One of Kotlin's key features is [null-safety](#). It deals with [null](#) values at compile time rather than deferring the problem to runtime and encountering a [NullPointerException](#). This helps to eliminate a common source of bugs without paying the cost of wrappers like [Optional](#). Kotlin also allows using functional constructs with nullable values as described in this [comprehensive guide to null-safety in Kotlin](#).

Although Java does not allow one to express null-safety in its type system, Spring Framework, Spring Data, and Reactor now provide null-safety of their API through tooling-friendly annotations. By default, types from Java APIs used in Kotlin are recognized as [platform types](#) for which null-checks are relaxed. [Kotlin's support for JSR 305 annotations](#) combined with nullability annotations provide null-safety for the related Spring API in Kotlin.

The JSR 305 checks can be configured by adding the `-Xjsr305` compiler flag with the following options: `-Xjsr305={strict|warn|ignore}`. The default behavior is the same as `-Xjsr305=warn`. The `strict` value is required to have null-safety taken in account in Kotlin types inferred from Spring API but should be used with the knowledge that Spring API nullability declaration could evolve even between minor releases and more checks may be added in the future).

WARNING Generic type arguments, varargs and array elements nullability are not yet supported. See [SPR-15942](#) for up-to-date information. Also be aware that Spring Boot's own API is [not yet annotated](#).

7.13.3. Kotlin API

runApplication

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

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

@SpringBootApplication
class MyApplication

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

This is a drop-in replacement for `SpringApplication.run(MyApplication::class.java, *args)`. It also allows customization of the application as shown in the following example:

```
runApplication<MyApplication>(*args) {
    setBannerMode(OFF)
}
```

Extensions

Kotlin [extensions](#) provide the ability to extend existing classes with additional functionality. The Spring Boot Kotlin API makes use of these extensions to add new Kotlin specific conveniences to existing APIs.

`TestRestTemplate` extensions, similar to those provided by Spring Framework for `RestOperations` in

Spring Framework, are provided. Among other things, the extensions make it possible to take advantage of Kotlin reified type parameters.

7.13.4. Dependency management

In order to avoid mixing different versions of Kotlin dependencies on the classpath, Spring Boot imports the Kotlin BOM.

With Maven, the Kotlin version can be customized by setting the `kotlin.version` property and plugin management is provided for `kotlin-maven-plugin`. With Gradle, the Spring Boot plugin automatically aligns the `kotlin.version` with the version of the Kotlin plugin.

Spring Boot also manages the version of Coroutines dependencies by importing the Kotlin Coroutines BOM. The version can be customized by setting the `kotlin-coroutines.version` property.

TIP `org.jetbrains.kotlinx:kotlinx-coroutines-reactor` dependency is provided by default if one bootstraps a Kotlin project with at least one reactive dependency on start.spring.io.

7.13.5. @ConfigurationProperties

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

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

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

TIP To generate `your own metadata` using the annotation processor, `kapt should be configured` with the `spring-boot-configuration-processor` dependency. Note that some features (such as detecting the default value or deprecated items) are not working due to limitations in the model `kapt` provides.

7.13.6. Testing

While it is possible to use JUnit 4 to test Kotlin code, JUnit 5 is provided by default and is recommended. JUnit 5 enables a test class to be instantiated once and reused for all of the class's tests. This makes it possible to use `@BeforeAll` and `@AfterAll` annotations on non-static methods, which is a good fit for Kotlin.

To mock Kotlin classes, [MockK](#) is recommended. If you need the [MockK](#) equivalent of the Mockito specific [@MockBean](#) and [@SpyBean](#) annotations, you can use [SpringMockK](#) which provides similar [@MockkBean](#) and [@SpykBean](#) annotations.

7.13.7. Resources

Further reading

- [Kotlin language reference](#)
- [Kotlin Slack](#) (with a dedicated #spring channel)
- [Stack Overflow with `spring` and `kotlin` tags](#)
- [Try Kotlin in your browser](#)
- [Kotlin blog](#)
- [Awesome Kotlin](#)
- [Tutorial: building web applications with Spring Boot and Kotlin](#)
- [Developing Spring Boot applications with Kotlin](#)
- [A Geospatial Messenger with Kotlin, Spring Boot and PostgreSQL](#)
- [Introducing Kotlin support in Spring Framework 5.0](#)
- [Spring Framework 5 Kotlin APIs, the functional way](#)

Examples

- [spring-boot-kotlin-demo](#): regular Spring Boot + Spring Data JPA project
- [mixit](#): Spring Boot 2 + WebFlux + Reactive Spring Data MongoDB
- [spring-kotlin-fullstack](#): WebFlux Kotlin fullstack example with Kotlin2js for frontend instead of JavaScript or TypeScript
- [spring-petclinic-kotlin](#): Kotlin version of the Spring PetClinic Sample Application
- [spring-kotlin-deepdive](#): a step by step migration for Boot 1.0 + Java to Boot 2.0 + Kotlin
- [spring-boot-coroutines-demo](#): Coroutines sample project

7.14. SSL

Spring Boot provides the ability to configure SSL trust material that can be applied to several types of connections in order to support secure communications. Configuration properties with the prefix [spring.ssl.bundle](#) can be used to specify named sets of trust material and associated information.

7.14.1. Configuring SSL With Java KeyStore Files

Configuration properties with the prefix [spring.ssl.bundle.jks](#) can be used to configure bundles of trust material created with the Java [keytool](#) utility and stored in Java KeyStore files in the JKS or PKCS12 format. Each bundle has a user-provided name that can be used to reference the bundle.

When used to secure an embedded web server, a [keystore](#) is typically configured with a Java

KeyStore containing a certificate and private key as shown in this example:

Properties

```
spring.ssl.bundle.jks.mybundle.key.alias=application  
spring.ssl.bundle.jks.mybundle.keystore.location=classpath:application.p12  
spring.ssl.bundle.jks.mybundle.keystore.password=secret  
spring.ssl.bundle.jks.mybundle.keystore.type=PKCS12
```

Yaml

```
spring:  
  ssl:  
    bundle:  
      jks:  
        mybundle:  
          key:  
            alias: "application"  
          keystore:  
            location: "classpath:application.p12"  
            password: "secret"  
            type: "PKCS12"
```

When used to secure a client-side connection, a `truststore` is typically configured with a Java KeyStore containing the server certificate as shown in this example:

Properties

```
spring.ssl.bundle.jks.mybundle.truststore.location=classpath:server.p12  
spring.ssl.bundle.jks.mybundle.truststore.password=secret
```

Yaml

```
spring:  
  ssl:  
    bundle:  
      jks:  
        mybundle:  
          truststore:  
            location: "classpath:server.p12"  
            password: "secret"
```

See [JksSslBundleProperties](#) for the full set of supported properties.

7.14.2. Configuring SSL With PEM-encoded Certificates

Configuration properties with the prefix `spring.ssl.bundle.pem` can be used to configure bundles of trust material in the form of PEM-encoded text. Each bundle has a user-provided name that can be

used to reference the bundle.

When used to secure an embedded web server, a **keystore** is typically configured with a certificate and private key as shown in this example:

Properties

```
spring.ssl.bundle.pem.mybundle.keystore.certificate=classpath:application.crt  
spring.ssl.bundle.pem.mybundle.keystore.private-key=classpath:application.key
```

Yaml

```
spring:  
  ssl:  
    bundle:  
      pem:  
        mybundle:  
          keystore:  
            certificate: "classpath:application.crt"  
            private-key: "classpath:application.key"
```

When used to secure a client-side connection, a **truststore** is typically configured with the server certificate as shown in this example:

Properties

```
spring.ssl.bundle.pem.mybundle.truststore.certificate=classpath:server.crt
```

Yaml

```
spring:  
  ssl:  
    bundle:  
      pem:  
        mybundle:  
          truststore:  
            certificate: "classpath:server.crt"
```

PEM content can be used directly for both the `certificate` and `private-key` properties. If the property values contains `BEGIN` and `END` markers then they will be treated as PEM content rather than a resource location.

The following example shows how a truststore certificate can be defined:

Properties

```
spring.ssl.bundle.pem.mybundle.truststore.certificate=\
-----BEGIN CERTIFICATE-----\n\
MIID1zCCAr+gAwIBAgIUNM5QQv8IzVQsgSmmdPQNaqyzWs4wDQYJKoZIhvcNAQEL\n\
BQAwezELMAkGA1UEBhMCWFgxEjAQBgNVBAgMCVN0YXR1TmFtZTERMA8GA1UEBwwI\n\
... \n\
V0IJjcmYjEZbTvpjFKznvaFiOUv+8L7jHQ1/Yf+9c3C8gSjdUfv88m17pqYXd+Ds\n\
HEmfNNjht130UyjNCITmLVXyy5p35vWmdf95U3uEbJSnNVtXH8qRmN9oK9mUpDb\n\
ngX6JBJI7fw7tXoqWSLHNiBODM88fUlQSho8\n\
-----END CERTIFICATE-----\n
```

Yaml

TIP

```
spring:
  ssl:
    bundle:
      pem:
        mybundle:
          truststore:
            certificate: |
              -----BEGIN CERTIFICATE-----\n
              MIID1zCCAr+gAwIBAgIUNM5QQv8IzVQsgSmmdPQNaqyzWs4wDQYJKoZIhvcNAQEL\n
              BQAwezELMAkGA1UEBhMCWFgxEjAQBgNVBAgMCVN0YXR1TmFtZTERMA8GA1UEBwwI\n
              ... \n
              V0IJjcmYjEZbTvpjFKznvaFiOUv+8L7jHQ1/Yf+9c3C8gSjdUfv88m17pqYXd+Ds\n
              HEmfNNjht130UyjNCITmLVXyy5p35vWmdf95U3uEbJSnNVtXH8qRmN9oK9mUpDb\n
              ngX6JBJI7fw7tXoqWSLHNiBODM88fUlQSho8\n
              -----END CERTIFICATE-----\n
```

See [PemSslBundleProperties](#) for the full set of supported properties.

7.14.3. Applying SSL Bundles

Once configured using properties, SSL bundles can be referred to by name in configuration properties for various types of connections that are auto-configured by Spring Boot. See the sections on [embedded web servers](#), [data technologies](#), and [REST clients](#) for further information.

7.14.4. Using SSL Bundles

Spring Boot auto-configures a bean of type `SslBundles` that provides access to each of the named bundles configured using the `spring.ssl.bundle` properties.

An `SslBundle` can be retrieved from the auto-configured `SslBundles` bean and used to create objects that are used to configure SSL connectivity in client libraries. The `SslBundle` provides a layered approach of obtaining these SSL objects:

- `getStores()` provides access to the key store and trust store `java.security.KeyStore` instances as well as any required key store password.
- `getManagers()` provides access to the `java.net.ssl.KeyManagerFactory` and `java.net.ssl.TrustManagerFactory` instances as well as the `java.net.ssl.KeyManager` and `java.net.ssl.TrustManager` arrays that they create.
- `createSslContext()` provides a convenient way to obtain a new `java.net.ssl.SSLContext` instance.

In addition, the `SslBundle` provides details about the key being used, the protocol to use and any option that should be applied to the SSL engine.

The following example shows retrieving an `SslBundle` and using it to create an `SSLContext`:

Java

```
import javax.net.ssl.SSLContext;

import org.springframework.boot.ssl.SslBundle;
import org.springframework.boot.ssl.SslBundles;
import org.springframework.stereotype.Component;

@Component
public class MyComponent {

    public MyComponent(SslBundles sslBundles) {
        SslBundle sslBundle = sslBundles.getBundle("mybundle");
        SSLContext sslContext = sslBundle.createSslContext();
        // do something with the created sslContext
    }

}
```

```
import org.springframework.boot.ssl.SslBundles
import org.springframework.stereotype.Component

@Component
class MyComponent(sslBundles: SslBundles) {

    init {
        val sslBundle = sslBundles.getBundle("mybundle")
        val sslContext = sslBundle.createSslContext()
        // do something with the created sslContext
    }

}
```

7.14.5. Reloading SSL bundles

SSL bundles can be reloaded when the key material changes. The component consuming the bundle has to be compatible with reloadable SSL bundles. Currently the following components are compatible:

- Tomcat web server
- Netty web server

To enable reloading, you need to opt-in via a configuration property as shown in this example:

Properties

```
spring.ssl.bundle.pem.mybundle.reload-on-update=true
spring.ssl.bundle.pem.mybundle.keystore.certificate=file:/some/directory/application.crt
spring.ssl.bundle.pem.mybundle.keystore.private-key-file:/some/directory/application.key
```

Yaml

```
spring:
  ssl:
    bundle:
      pem:
        mybundle:
          reload-on-update: true
          keystore:
            certificate: "file:/some/directory/application.crt"
            private-key: "file:/some/directory/application.key"
```

A file watcher is then watching the files and if they change, the SSL bundle will be reloaded. This in

turn triggers a reload in the consuming component, e.g. Tomcat rotates the certificates in the SSL enabled connectors.

You can configure the quiet period (to make sure that there are no more changes) of the file watcher with the `spring.ssl.bundle.watch.file.quiet-period` property.

7.15. What to Read Next

If you want to learn more about any of the classes discussed in this section, see the [Spring Boot API documentation](#) or you can browse the [source code directly](#). If you have specific questions, see the [how-to section](#).

If you are comfortable with Spring Boot's core features, you can continue on and read about [production-ready features](#).

Chapter 8. Web

Spring Boot is well suited for web application development. You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the `spring-boot-starter-web` module to get up and running quickly. You can also choose to build reactive web applications by using the `spring-boot-starter-webflux` module.

If you have not yet developed a Spring Boot web application, you can follow the "Hello World!" example in the [Getting started](#) section.

8.1. Servlet Web Applications

If you want to build servlet-based web applications, you can take advantage of Spring Boot's auto-configuration for Spring MVC or Jersey.

8.1.1. The “Spring Web MVC Framework”

The [Spring Web MVC framework](#) (often referred to as “Spring MVC”) is a rich “model view controller” web framework. Spring MVC lets you create special `@Controller` or `@RestController` beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using `@RequestMapping` annotations.

The following code shows a typical `@RestController` that serves JSON data:

Java

```
import java.util.List;

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class MyRestController {

    private final UserRepository userRepository;
    private final CustomerRepository customerRepository;

    public MyRestController(UserRepository userRepository, CustomerRepository
customerRepository) {
        this.userRepository = userRepository;
        this.customerRepository = customerRepository;
    }

    @GetMapping("/{userId}")
    public User getUser(@PathVariable Long userId) {
        return this.userRepository.findById(userId).get();
    }

    @GetMapping("/{userId}/customers")
    public List<Customer> getUserCustomers(@PathVariable Long userId) {
        return
this.userRepository.findById(userId).map(this.customerRepository::findByUser).get();
    }

    @DeleteMapping("/{userId}")
    public void deleteUser(@PathVariable Long userId) {
        this.userRepository.deleteById(userId);
    }

}
```

Kotlin

```
import org.springframework.web.bind.annotation.DeleteMapping
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@RestController
@RequestMapping("/users")
class MyRestController(private val userRepository: UserRepository, private val
customerRepository: CustomerRepository) {

    @GetMapping("/{userId}")
    fun getUser(@PathVariable userId: Long): User {
        return userRepository.findById(userId).get()
    }

    @GetMapping("/{userId}/customers")
    fun getUserCustomers(@PathVariable userId: Long): List<Customer> {
        return
userRepository.findById(userId).map(customerRepository::findByUser).get()
    }

    @DeleteMapping("/{userId}")
    fun deleteUser(@PathVariable userId: Long) {
        userRepository.deleteById(userId)
    }

}
```

“WebMvc.fn”, the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import org.springframework.web.servlet.function.RequestPredicate;
import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.ServerResponse;

import static org.springframework.web.servlet.function.RequestPredicates.accept;
import static org.springframework.web.servlet.function.RouterFunctions.route;

@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {

    private static final RequestPredicate ACCEPT_JSON =
accept(MediaType.APPLICATION_JSON);

    @Bean
    public RouterFunction<ServerResponse> routerFunction(MyUserHandler userHandler) {
        return route()
            .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
            .GET("/{user}/customers", ACCEPT_JSON, userHandler::getUserCustomers)
            .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
            .build();
    }

}
```

Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.MediaType
import org.springframework.web.servlet.function.RequestPredicates.accept
import org.springframework.web.servlet.function.RouterFunction
import org.springframework.web.servlet.function.RouterFunctions
import org.springframework.web.servlet.function.ServerResponse

@Configuration(proxyBeanMethods = false)
class MyRoutingConfiguration {

    @Bean
    fun routerFunction(userHandler: MyUserHandler): RouterFunction<ServerResponse> {
        return RouterFunctions.route()
            .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
            .GET("/{user}/customers", ACCEPT_JSON, userHandler::getUserCustomers)
            .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
            .build()
    }

    companion object {
        private val ACCEPT_JSON = accept(MediaType.APPLICATION_JSON)
    }
}
```

Java

```
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.function.ServerRequest;
import org.springframework.web.servlet.function.ServerResponse;

@Component
public class MyUserHandler {

    public ServerResponse getUser(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

    public ServerResponse getUserCustomers(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

    public ServerResponse deleteUser(ServerRequest request) {
        ...
        return ServerResponse.ok().build();
    }

}
```

Kotlin

```
import org.springframework.stereotype.Component
import org.springframework.web.servlet.function.ServerRequest
import org.springframework.web.servlet.function.ServerResponse

@Component
class MyUserHandler {

    fun getUser(request: ServerRequest?): ServerResponse {
        return ServerResponse.ok().build()
    }

    fun getUserCustomers(request: ServerRequest?): ServerResponse {
        return ServerResponse.ok().build()
    }

    fun deleteUser(request: ServerRequest?): ServerResponse {
        return ServerResponse.ok().build()
    }

}
```

Spring MVC is part of the core Spring Framework, and detailed information is available in the

reference documentation. There are also several guides that cover Spring MVC available at spring.io/guides.

TIP You can define as many `RouterFunction` beans as you like to modularize the definition of the router. Beans can be ordered if you need to apply a precedence.

Spring MVC Auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications. It replaces the need for `@EnableWebMvc` and the two cannot be used together. In addition to Spring MVC's defaults, the auto-configuration provides the following features:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered [later in this document](#)).
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
- Support for `HttpMessageConverters` (covered [later in this document](#)).
- Automatic registration of `MessageCodesResolver` (covered [later in this document](#)).
- Static `index.html` support.
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered [later in this document](#)).

If you want to keep those Spring Boot MVC customizations and make more [MVC customizations](#) (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`.

If you want to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, and still keep the Spring Boot MVC customizations, you can declare a bean of type `WebMvcRegistrations` and use it to provide custom instances of those components. The custom instances will be subject to further initialization and configuration by Spring MVC. To participate in, and if desired, override that subsequent processing, a `WebMvcConfigurer` should be used.

If you do not want to use the auto-configuration and want to take complete control of Spring MVC, add your own `@Configuration` annotated with `@EnableWebMvc`. Alternatively, add your own `@Configuration`-annotated `DelegatingWebMvcConfiguration` as described in the Javadoc of `@EnableWebMvc`.

Spring MVC Conversion Service

Spring MVC uses a different `ConversionService` to the one used to convert values from your `application.properties` or `application.yaml` file. It means that `Period`, `Duration` and `DataSize` converters are not available and that `@DurationUnit` and `@DataSizeUnit` annotations will be ignored.

If you want to customize the `ConversionService` used by Spring MVC, you can provide a `WebMvcConfigurer` bean with an `addFormatters` method. From this method you can register any converter that you like, or you can delegate to the static methods available on `ApplicationConversionService`.

Conversion can also be customized using the `spring.mvc.format.*` configuration properties. When not configured, the following defaults are used:

Property	DateFormatter
<code>spring.mvc.format.date</code>	<code>ofLocalizedDate(FormatStyle.SHORT)</code>
<code>spring.mvc.format.time</code>	<code>ofLocalizedTime(FormatStyle.SHORT)</code>
<code>spring.mvc.format.date-time</code>	<code>ofLocalDateTime(FormatStyle.SHORT)</code>

HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box. For example, objects can be automatically converted to JSON (by using the Jackson library) or XML (by using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in `UTF-8`.

Any `HttpMessageConverter` bean that is present in the context is added to the list of converters. You can also override default converters in the same way.

If you need to add or customize converters, you can use Spring Boot's `HttpMessageConverters` class, as shown in the following listing:

Java

```
import org.springframework.boot.autoconfigure.http.HttpMessageConverters;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.converter.HttpMessageConverter;

@Configuration(proxyBeanMethods = false)
public class MyHttpMessageConvertersConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = new AdditionalHttpMessageConverter();
        HttpMessageConverter<?> another = new AnotherHttpMessageConverter();
        return new HttpMessageConverters(additional, another);
    }

}
```

```

import org.springframework.boot.autoconfigure.http.HttpMessageConverters
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.converter.HttpMessageConverter

@Configuration(proxyBeanMethods = false)
class MyHttpMessageConvertersConfiguration {

    @Bean
    fun customConverters(): HttpMessageConverters {
        val additional: HttpMessageConverter<*> = AdditionalHttpMessageConverter()
        val another: HttpMessageConverter<*> = AnotherHttpMessageConverter()
        return HttpMessageConverters(additional, another)
    }

}

```

For further control, you can also sub-class `HttpMessageConverters` and override its `postProcessConverters` and/or `postProcessPartConverters` methods. This can be useful when you want to re-order or remove some of the converters that Spring MVC configures by default.

MessageCodesResolver

Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: `MessageCodesResolver`. If you set the `spring.mvc.message-codes-resolver-format` property `PREFIX_ERROR_CODE` or `POSTFIX_ERROR_CODE`, Spring Boot creates one for you (see the enumeration in `DefaultMessageCodesResolver.Format`).

Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath or from the root of the `ServletContext`. It uses the `ResourceHttpRequestHandler` from Spring MVC so that you can modify that behavior by adding your own `WebMvcConfigurer` and overriding the `addResourceHandlers` method.

In a stand-alone web application, the default servlet from the container is not enabled. It can be enabled using the `server.servlet.register-default-servlet` property.

The default servlet acts as a fallback, serving content from the root of the `ServletContext` if Spring decides not to handle it. Most of the time, this does not happen (unless you modify the default MVC configuration), because Spring can always handle requests through the `DispatcherServlet`.

By default, resources are mapped on `/**`, but you can tune that with the `spring.mvc.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

Properties

```
spring.mvc.static-path-pattern=/resources/**
```

Yaml

```
spring:  
  mvc:  
    static-path-pattern: "/resources/**"
```

You can also customize the static resource locations by using the `spring.web.resources.static-locations` property (replacing the default values with a list of directory locations). The root servlet context path, `"/"`, is automatically added as a location as well.

In addition to the “standard” static resource locations mentioned earlier, a special case is made for [Webjars content](#). By default, any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format. The path can be customized with the `spring.mvc.webjars-path-pattern` property.

Do not use the `src/main/webapp` directory if your application is packaged as a jar.

TIP Although this directory is a common standard, it works **only** with war packaging, and it is silently ignored by most build tools if you generate a jar.

Spring Boot also supports the advanced resource handling features provided by Spring MVC, allowing use cases such as cache-busting static resources or using version agnostic URLs for Webjars.

To use version agnostic URLs for Webjars, add the `webjars-locator-core` dependency. Then declare your Webjar. Using jQuery as an example, adding `"/webjars/jquery/jquery.min.js"` results in `"/webjars/jquery/x.y.z/jquery.min.js"` where `x.y.z` is the Webjar version.

NOTE If you use JBoss, you need to declare the `webjars-locator-jboss-vfs` dependency instead of the `webjars-locator-core`. Otherwise, all Webjars resolve as a `404`.

To use cache busting, the following configuration configures a cache busting solution for all static resources, effectively adding a content hash, such as `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`, in URLs:

Properties

```
spring.web.resources.chain.strategy.content.enabled=true  
spring.web.resources.chain.strategy.content.paths=/**
```

Yaml

```
spring:  
  web:  
    resources:  
      chain:  
        strategy:  
          content:  
            enabled: true  
            paths: "/**"
```

NOTE Links to resources are rewritten in templates at runtime, thanks to a [ResourceUrlEncodingFilter](#) that is auto-configured for Thymeleaf and FreeMarker. You should manually declare this filter when using JSPs. Other template engines are currently not automatically supported but can be with custom template macros/helpers and the use of the [ResourceUrlProvider](#).

When loading resources dynamically with, for example, a JavaScript module loader, renaming files is not an option. That is why other strategies are also supported and can be combined. A "fixed" strategy adds a static version string in the URL without changing the file name, as shown in the following example:

Properties

```
spring.web.resources.chain.strategy.content.enabled=true  
spring.web.resources.chain.strategy.content.paths=**  
spring.web.resources.chain.strategy.fixed.enabled=true  
spring.web.resources.chain.strategy.fixed.paths=/js/lib/  
spring.web.resources.chain.strategy.fixed.version=v12
```

Yaml

```
spring:  
  web:  
    resources:  
      chain:  
        strategy:  
          content:  
            enabled: true  
            paths: "/**"  
        fixed:  
          enabled: true  
          paths: "/js/lib/"  
          version: "v12"
```

With this configuration, JavaScript modules located under ["/js/lib/"](#) use a fixed versioning strategy (["/v12/js/lib/mymodule.js"](#)), while other resources still use the content one ([`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`](#)).

See [WebProperties.Resources](#) for more supported options.

TIP This feature has been thoroughly described in a dedicated [blog post](#) and in Spring Framework's [reference documentation](#).

Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

This only acts as a fallback for actual index routes defined by the application. The ordering is defined by the order of `HandlerMapping` beans which is by default the following:

<code>RouterFunctionMapping</code>	Endpoints declared with <code>RouterFunction</code> beans
<code>RequestMappingHandlerMapping</code>	Endpoints declared in <code>@Controller</code> beans
<code>WelcomePageHandlerMapping</code>	The welcome page support

Custom Favicon

As with other static resources, Spring Boot checks for a `favicon.ico` in the configured static content locations. If such a file is present, it is automatically used as the favicon of the application.

Path Matching and Content Negotiation

Spring MVC can map incoming HTTP requests to handlers by looking at the request path and matching it to the mappings defined in your application (for example, `@GetMapping` annotations on Controller methods).

Spring Boot chooses to disable suffix pattern matching by default, which means that requests like "`GET /projects/spring-boot.json`" will not be matched to `@GetMapping("/projects/spring-boot")` mappings. This is considered as a [best practice for Spring MVC applications](#). This feature was mainly useful in the past for HTTP clients which did not send proper "Accept" request headers; we needed to make sure to send the correct Content Type to the client. Nowadays, Content Negotiation is much more reliable.

There are other ways to deal with HTTP clients that do not consistently send proper "Accept" request headers. Instead of using suffix matching, we can use a query parameter to ensure that requests like "`GET /projects/spring-boot?format=json`" will be mapped to `@GetMapping("/projects/spring-boot")`:

Properties

```
spring.mvc.contentnegotiation.favor-parameter=true
```

Yaml

```
spring:  
  mvc:  
    contentnegotiation:  
      favor-parameter: true
```

Or if you prefer to use a different parameter name:

Properties

```
spring.mvc.contentnegotiation.favor-parameter=true  
spring.mvc.contentnegotiation.parameter-name=myparam
```

Yaml

```
spring:  
  mvc:  
    contentnegotiation:  
      favor-parameter: true  
      parameter-name: "myparam"
```

Most standard media types are supported out-of-the-box, but you can also define new ones:

Properties

```
spring.mvc.contentnegotiation.media-types.markdown=text/markdown
```

Yaml

```
spring:  
  mvc:  
    contentnegotiation:  
      media-types:  
        markdown: "text/markdown"
```

As of Spring Framework 5.3, Spring MVC supports two strategies for matching request paths to controllers. By default, Spring Boot uses the [PathPatternParser](#) strategy. [PathPatternParser](#) is an [optimized implementation](#) but comes with some restrictions compared to the [AntPathMatcher](#) strategy. [PathPatternParser](#) restricts usage of [some path pattern variants](#). It is also incompatible with configuring the [DispatcherServlet](#) with a path prefix ([spring.mvc.servlet.path](#)).

The strategy can be configured using the [spring.mvc.pathmatch.matching-strategy](#) configuration property, as shown in the following example:

Properties

```
spring.mvc.pathmatch.matching-strategy=ant-path-matcher
```

Yaml

```
spring:  
  mvc:  
    pathmatch:  
      matching-strategy: "ant-path-matcher"
```

By default, Spring MVC will send a 404 Not Found error response if a handler is not found for a request. To have a `NoHandlerFoundException` thrown instead, set `configprop:spring.mvc.throw-exception-if-no-handler-found` to `true`. Note that, by default, the `serving of static content` is mapped to `/**` and will, therefore, provide a handler for all requests. For a `NoHandlerFoundException` to be thrown, you must also set `spring.mvc.static-path-pattern` to a more specific value such as `/resources/**` or set `spring.web.resources.add-mappings` to `false` to disable serving of static content entirely.

ConfigurableWebBindingInitializer

Spring MVC uses a `WebBindingInitializer` to initialize a `WebDataBinder` for a particular request. If you create your own `ConfigurableWebBindingInitializer @Bean`, Spring Boot automatically configures Spring MVC to use it.

Template Engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

- [FreeMarker](#)
- [Groovy](#)
- [Thymeleaf](#)
- [Mustache](#)

TIP

If possible, JSPs should be avoided. There are several [known limitations](#) when using them with embedded servlet containers.

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

TIP

Depending on how you run your application, your IDE may order the classpath differently. Running your application in the IDE from its main method results in a different ordering than when you run your application by using Maven or Gradle or from its packaged jar. This can cause Spring Boot to fail to find the expected template. If you have this problem, you can reorder the classpath in the IDE to place the module's classes and resources first.

Error Handling

By default, Spring Boot provides an `/error` mapping that handles all errors in a sensible way, and it is registered as a “global” error page in the servlet container. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a “whitelabel” error view that renders the same data in HTML format (to customize it, add a `View` that resolves to `error`).

There are a number of `server.error` properties that can be set if you want to customize the default error handling behavior. See the “[Server Properties](#)” section of the Appendix.

To replace the default behavior completely, you can implement `ErrorController` and register a bean definition of that type or add a bean of type `ErrorAttributes` to use the existing mechanism but replace the contents.

TIP

The `BasicErrorController` can be used as a base class for a custom `ErrorController`. This is particularly useful if you want to add a handler for a new content type (the default is to handle `text/html` specifically and provide a fallback for everything else). To do so, extend `BasicErrorController`, add a public method with a `@RequestMapping` that has a `produces` attribute, and create a bean of your new type.

As of Spring Framework 6.0, [RFC 7807 Problem Details](#) is supported. Spring MVC can produce custom error messages with the `application/problem+json` media type, like:

```
{  
  "type": "https://example.org/problems/unknown-project",  
  "title": "Unknown project",  
  "status": 404,  
  "detail": "No project found for id 'spring-unknown'",  
  "instance": "/projects/spring-unknown"  
}
```

This support can be enabled by setting `spring.mvc.problemdetails.enabled` to `true`.

You can also define a class annotated with `@ControllerAdvice` to customize the JSON document to return for a particular controller and/or exception type, as shown in the following example:

Java

```
import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.http.HttpServletRequest;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

@ControllerAdvice(basePackageClasses = SomeController.class)
public class MyControllerAdvice extends ResponseEntityExceptionHandler {

    @ResponseBody
    @ExceptionHandler(MyException.class)
    public ResponseEntity<?> handleControllerException(HttpServletRequest request,
Throwable ex) {
        HttpStatus status = getStatus(request);
        return new ResponseEntity<>(new MyErrorResponse(status.value(), ex.getMessage()),
status);
    }

    private HttpStatus getStatus(HttpServletRequest request) {
        Integer code = (Integer)
request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
        HttpStatus status = HttpStatus.resolve(code);
        return (status != null) ? status : HttpStatus.INTERNAL_SERVER_ERROR;
    }
}
```

```

import jakarta.servlet.RequestDispatcher
import jakarta.servlet.http.HttpServletRequest
import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.ControllerAdvice
import org.springframework.web.bind.annotation.ExceptionHandler
import org.springframework.web.bind.annotation.ResponseBody
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler

@ControllerAdvice(basePackageClasses = [SomeController::class])
class MyControllerAdvice : ResponseEntityExceptionHandler() {

    @ResponseBody
    @ExceptionHandler(MyException::class)
    fun handleControllerException(request: HttpServletRequest, ex: Throwable):
    ResponseEntity<*> {
        val status = getStatus(request)
        return ResponseEntity(MyErrorResponse(status.value(), ex.message), status)
    }

    private fun getStatus(request: HttpServletRequest): HttpStatus {
        val code = request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE) as Int
        val status = HttpStatus.resolve(code)
        return status ?: HttpStatus.INTERNAL_SERVER_ERROR
    }
}

```

In the preceding example, if `MyException` is thrown by a controller defined in the same package as `SomeController`, a JSON representation of the `MyErrorResponse` POJO is used instead of the `ErrorAttributes` representation.

In some cases, errors handled at the controller level are not recorded by web observations or the [metrics infrastructure](#). Applications can ensure that such exceptions are recorded with the observations by [setting the handled exception on the observation context](#).

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` directory. Error pages can either be static HTML (that is, added under any of the static resource directories) or be built by using templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your directory structure would be as follows:

```
src/
+- main/
  +- java/
    |   + <source code>
  +- resources/
    +- public/
      +- error/
        |   +- 404.html
      +- <other public assets>
```

To map all `5xx` errors by using a FreeMarker template, your directory structure would be as follows:

```
src/
+- main/
  +- java/
    |   + <source code>
  +- resources/
    +- templates/
      +- error/
        |   +- 5xx.ftlh
      +- <other templates>
```

For more complex mappings, you can also add beans that implement the `ErrorViewResolver` interface, as shown in the following example:

Java

```
import java.util.Map;

import jakarta.servlet.http.HttpServletRequest;

import org.springframework.boot.autoconfigure.web.servlet.error.ErrorViewResolver;
import org.springframework.http.HttpStatus;
import org.springframework.web.servlet.ModelAndView;

public class MyErrorViewResolver implements ErrorViewResolver {

    @Override
    public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus
status, Map<String, Object> model) {
        // Use the request or status to optionally return a ModelAndView
        if (status == HttpStatus.INSUFFICIENT_STORAGE) {
            // We could add custom model values here
            new ModelAndView("myview");
        }
        return null;
    }

}
```

Kotlin

```
import jakarta.servlet.http.HttpServletRequest
import org.springframework.boot.autoconfigure.web.servlet.error.ErrorViewResolver
import org.springframework.http.HttpStatus
import org.springframework.web.servlet.ModelAndView

class MyErrorViewResolver : ErrorViewResolver {

    override fun resolveErrorView(request: HttpServletRequest, status: HttpStatus,
        model: Map<String, Any>): ModelAndView? {
        // Use the request or status to optionally return a ModelAndView
        if (status == HttpStatus.INSUFFICIENT_STORAGE) {
            // We could add custom model values here
            return ModelAndView("myview")
        }
        return null
    }

}
```

You can also use regular Spring MVC features such as `@ExceptionHandler` methods and `@ControllerAdvice`. The `ErrorController` then picks up any unhandled exceptions.

Mapping Error Pages Outside of Spring MVC

For applications that do not use Spring MVC, you can use the `ErrorPageRegistrar` interface to directly register `ErrorPages`. This abstraction works directly with the underlying embedded servlet container and works even if you do not have a Spring MVC `DispatcherServlet`.

Java

```
import org.springframework.boot.web.server.ErrorPage;
import org.springframework.boot.web.server.ErrorPageRegistrar;
import org.springframework.boot.web.server.ErrorPageRegistry;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpStatus;

@Configuration(proxyBeanMethods = false)
public class MyErrorPagesConfiguration {

    @Bean
    public ErrorPageRegistrar errorPageRegistrar() {
        return this::registerErrorPages;
    }

    private void registerErrorPages(ErrorPageRegistry registry) {
        registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    }

}
```

```
import org.springframework.boot.web.server.ErrorPage
import org.springframework.boot.web.server.ErrorPageRegistrar
import org.springframework.boot.web.server.ErrorPageRegistry
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.HttpStatus

@Configuration(proxyBeanMethods = false)
class MyErrorPagesConfiguration {

    @Bean
    fun errorPageRegistrar(): ErrorPageRegistrar {
        return ErrorPageRegistrar { registry: ErrorPageRegistry ->
            registerErrorPages(registry)
    }

    private fun registerErrorPages(registry: ErrorPageRegistry) {
        registry.addErrorPages(ErrorPage(HttpStatus.BAD_REQUEST, "/400"))
    }
}
```

NOTE

If you register an `ErrorPage` with a path that ends up being handled by a `Filter` (as is common with some non-Spring web frameworks, like Jersey and Wicket), then the `Filter` has to be explicitly registered as an `ERROR` dispatcher, as shown in the following example:

Java

```
import java.util.EnumSet;

import jakarta.servlet.DispatcherType;

import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyFilterConfiguration {

    @Bean
    public FilterRegistrationBean<MyFilter> myFilter() {
        FilterRegistrationBean<MyFilter> registration = new
FilterRegistrationBean<>(new MyFilter());
        // ...
        registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.class));
        return registration;
    }

}
```

Kotlin

```
import jakarta.servlet.DispatcherType
import org.springframework.boot.web.servlet.FilterRegistrationBean
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import java.util.EnumSet

@Configuration(proxyBeanMethods = false)
class MyFilterConfiguration {

    @Bean
    fun myFilter(): FilterRegistrationBean<MyFilter> {
        val registration = FilterRegistrationBean(MyFilter())
        // ...
        registration.setDispatcherTypes(EnumSet.allOf(DispatcherType::class.java))
        return registration
    }

}
```

Note that the default `FilterRegistrationBean` does not include the `ERROR` dispatcher type.

Error Handling in a WAR Deployment

When deployed to a servlet container, Spring Boot uses its error page filter to forward a request

with an error status to the appropriate error page. This is necessary as the servlet specification does not provide an API for registering error pages. Depending on the container that you are deploying your war file to and the technologies that your application uses, some additional configuration may be required.

The error page filter can only forward the request to the correct error page if the response has not already been committed. By default, WebSphere Application Server 8.0 and later commits the response upon successful completion of a servlet's service method. You should disable this behavior by setting `com.ibm.ws.webcontainer.invokeFlushAfterService` to `false`.

CORS Support

[Cross-origin resource sharing](#) (CORS) is a [W3C specification](#) implemented by [most browsers](#) that lets you specify in a flexible way what kind of cross-domain requests are authorized, instead of using some less secure and less powerful approaches such as IFRAAME or JSONP.

As of version 4.2, Spring MVC [supports CORS](#). Using [controller method CORS configuration](#) with `@CrossOrigin` annotations in your Spring Boot application does not require any specific configuration. [Global CORS configuration](#) can be defined by registering a `WebMvcConfigurer` bean with a customized `addCorsMappings(CorsRegistry)` method, as shown in the following example:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration(proxyBeanMethods = false)
public class MyCorsConfiguration {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {

            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**");
            }
        };
    }
}
```

```

import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.web.servlet.config.annotation.CorsRegistry
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Configuration(proxyBeanMethods = false)
class MyCorsConfiguration {

    @Bean
    fun corsConfigurer(): WebMvcConfigurer {
        return object : WebMvcConfigurer {
            override fun addCorsMappings(registry: CorsRegistry) {
                registry.addMapping("/api/**")
            }
        }
    }
}

```

8.1.2. JAX-RS and Jersey

If you prefer the JAX-RS programming model for REST endpoints, you can use one of the available implementations instead of Spring MVC. [Jersey](#) and [Apache CXF](#) work quite well out of the box. CXF requires you to register its [Servlet](#) or [Filter](#) as a [@Bean](#) in your application context. Jersey has some native Spring support, so we also provide auto-configuration support for it in Spring Boot, together with a starter.

To get started with Jersey, include the [spring-boot-starter-jersey](#) as a dependency and then you need one [@Bean](#) of type [ResourceConfig](#) in which you register all the endpoints, as shown in the following example:

```

import org.glassfish.jersey.server.ResourceConfig;

import org.springframework.stereotype.Component;

@Component
public class MyJerseyConfig extends ResourceConfig {

    public MyJerseyConfig() {
        register(MyEndpoint.class);
    }

}

```

WARNING

Jersey's support for scanning executable archives is rather limited. For example, it cannot scan for endpoints in a package found in a [fully executable jar file](#) or in [WEB-INF/classes](#) when running an executable war file. To avoid this limitation, the `packages` method should not be used, and endpoints should be registered individually by using the `register` method, as shown in the preceding example.

For more advanced customizations, you can also register an arbitrary number of beans that implement `ResourceConfigCustomizer`.

All the registered endpoints should be `@Components` with HTTP resource annotations (`@GET` and others), as shown in the following example:

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

import org.springframework.stereotype.Component;

@Component
@Path("/hello")
public class MyEndpoint {

    @GET
    public String message() {
        return "Hello";
    }

}
```

Since the `Endpoint` is a Spring `@Component`, its lifecycle is managed by Spring and you can use the `@Autowired` annotation to inject dependencies and use the `@Value` annotation to inject external configuration. By default, the Jersey servlet is registered and mapped to `/*`. You can change the mapping by adding `@ApplicationPath` to your `ResourceConfig`.

By default, Jersey is set up as a servlet in a `@Bean` of type `ServletRegistrationBean` named `jerseyServletRegistration`. By default, the servlet is initialized lazily, but you can customize that behavior by setting `spring.jersey.servlet.load-on-startup`. You can disable or override that bean by creating one of your own with the same name. You can also use a filter instead of a servlet by setting `spring.jersey.type=filter` (in which case, the `@Bean` to replace or override is `jerseyFilterRegistration`). The filter has an `@Order`, which you can set with `spring.jersey.filter.order`. When using Jersey as a filter, a servlet that will handle any requests that are not intercepted by Jersey must be present. If your application does not contain such a servlet, you may want to enable the default servlet by setting `server.servlet.register-default-servlet` to `true`. Both the servlet and the filter registrations can be given init parameters by using `spring.jersey.init.*` to specify a map of properties.

8.1.3. Embedded Servlet Container Support

For servlet application, Spring Boot includes support for embedded [Tomcat](#), [Jetty](#), and [Undertow](#) servers. Most developers use the appropriate “Starter” to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port [8080](#).

Servlets, Filters, and Listeners

When using an embedded servlet container, you can register servlets, filters, and all the listeners (such as [HttpSessionListener](#)) from the servlet spec, either by using Spring beans or by scanning for servlet components.

Registering Servlets, Filters, and Listeners as Spring Beans

Any [Servlet](#), [Filter](#), or servlet [*Listener](#) instance that is a Spring bean is registered with the embedded container. This can be particularly convenient if you want to refer to a value from your [application.properties](#) during configuration.

By default, if the context contains only a single [Servlet](#), it is mapped to [/](#). In the case of multiple [servlet beans](#), the bean name is used as a path prefix. [Filters](#) map to [/*](#).

If convention-based mapping is not flexible enough, you can use the [ServletRegistrationBean](#), [FilterRegistrationBean](#), and [ServletListenerRegistrationBean](#) classes for complete control.

It is usually safe to leave filter beans unordered. If a specific order is required, you should annotate the [Filter](#) with [@Order](#) or make it implement [Ordered](#). You cannot configure the order of a [Filter](#) by annotating its bean method with [@Order](#). If you cannot change the [Filter](#) class to add [@Order](#) or implement [Ordered](#), you must define a [FilterRegistrationBean](#) for the [Filter](#) and set the registration bean’s order using the [setOrder\(int\)](#) method. Avoid configuring a filter that reads the request body at [Ordered.HIGHEST_PRECEDENCE](#), since it might go against the character encoding configuration of your application. If a servlet filter wraps the request, it should be configured with an order that is less than or equal to [OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER](#).

TIP To see the order of every [Filter](#) in your application, enable debug level logging for the [web logging group](#) ([logging.level.web=debug](#)). Details of the registered filters, including their order and URL patterns, will then be logged at startup.

WARNING Take care when registering [Filter](#) beans since they are initialized very early in the application lifecycle. If you need to register a [Filter](#) that interacts with other beans, consider using a [DelegatingFilterProxyRegistrationBean](#) instead.

Servlet Context Initialization

Embedded servlet containers do not directly execute the [jakarta.servlet.ServletContainerInitializer](#) interface or Spring’s [org.springframework.web.WebApplicationInitializer](#) interface. This is an intentional design decision intended to reduce the risk that third party libraries designed to run inside a war may break Spring Boot applications.

If you need to perform servlet context initialization in a Spring Boot application, you should register a bean that implements the `org.springframework.boot.web.servlet.ServletContextInitializer` interface. The single `onStartup` method provides access to the `ServletContext` and, if necessary, can easily be used as an adapter to an existing `WebApplicationInitializer`.

Scanning for Servlets, Filters, and listeners

When using an embedded container, automatic registration of classes annotated with `@WebServlet`, `@WebFilter`, and `@WebListener` can be enabled by using `@ServletComponentScan`.

TIP `@ServletComponentScan` has no effect in a standalone container, where the container's built-in discovery mechanisms are used instead.

The `ServletWebServerApplicationContext`

Under the hood, Spring Boot uses a different type of `ApplicationContext` for embedded servlet container support. The `ServletWebServerApplicationContext` is a special type of `WebApplicationContext` that bootstraps itself by searching for a single `ServletWebServerFactory` bean. Usually a `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` has been auto-configured.

NOTE You usually do not need to be aware of these implementation classes. Most applications are auto-configured, and the appropriate `ApplicationContext` and `ServletWebServerFactory` are created on your behalf.

In an embedded container setup, the `ServletContext` is set as part of server startup which happens during application context initialization. Because of this beans in the `ApplicationContext` cannot be reliably initialized with a `ServletContext`. One way to get around this is to inject `ApplicationContext` as a dependency of the bean and access the `ServletContext` only when it is needed. Another way is to use a callback once the server has started. This can be done using an `ApplicationListener` which listens for the `ApplicationStartedEvent` as follows:

```

import jakarta.servlet.ServletContext;

import org.springframework.boot.context.event.ApplicationStartedEvent;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationListener;
import org.springframework.web.context.WebApplicationContext;

public class MyDemoBean implements ApplicationListener<ApplicationStartedEvent> {

    private ServletContext servletContext;

    @Override
    public void onApplicationEvent(ApplicationStartedEvent event) {
        ApplicationContext applicationContext = event.getApplicationContext();
        this.servletContext = ((WebApplicationContext)
applicationContext).getServletContext();
    }

}

```

Customizing Embedded Servlet Containers

Common servlet container settings can be configured by using Spring [Environment](#) properties. Usually, you would define the properties in your [application.properties](#) or [application.yaml](#) file.

Common server settings include:

- Network settings: Listen port for incoming HTTP requests ([server.port](#)), interface address to bind to ([server.address](#)), and so on.
- Session settings: Whether the session is persistent ([server.servlet.session.persistent](#)), session timeout ([server.servlet.session.timeout](#)), location of session data ([server.servlet.session.store-dir](#)), and session-cookie configuration ([server.servlet.session.cookie.*](#)).
- Error management: Location of the error page ([server.error.path](#)) and so on.
- [SSL](#)
- [HTTP compression](#)

Spring Boot tries as much as possible to expose common settings, but this is not always possible. For those cases, dedicated namespaces offer server-specific customizations (see [server.tomcat](#) and [server.undertow](#)). For instance, [access logs](#) can be configured with specific features of the embedded servlet container.

TIP See the [ServerProperties](#) class for a complete list.

SameSite Cookies

The [SameSite](#) cookie attribute can be used by web browsers to control if and how cookies are

submitted in cross-site requests. The attribute is particularly relevant for modern web browsers which have started to change the default value that is used when the attribute is missing.

If you want to change the `SameSite` attribute of your session cookie, you can use the `server.servlet.session.cookie.same-site` property. This property is supported by auto-configured Tomcat, Jetty and Undertow servers. It is also used to configure Spring Session servlet based `SessionRepository` beans.

For example, if you want your session cookie to have a `SameSite` attribute of `None`, you can add the following to your `application.properties` or `application.yaml` file:

Properties

```
server.servlet.session.cookie.same-site=none
```

Yaml

```
server:
  servlet:
    session:
      cookie:
        same-site: "none"
```

If you want to change the `SameSite` attribute on other cookies added to your `HttpServletResponse`, you can use a `CookieSameSiteSupplier`. The `CookieSameSiteSupplier` is passed a `Cookie` and may return a `SameSite` value, or `null`.

There are a number of convenience factory and filter methods that you can use to quickly match specific cookies. For example, adding the following bean will automatically apply a `SameSite` of `Lax` for all cookies with a name that matches the regular expression `myapp.*`.

Java

```
import org.springframework.boot.web.servlet.server.CookieSameSiteSupplier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MySameSiteConfiguration {

    @Bean
    public CookieSameSiteSupplier applicationCookieSameSiteSupplier() {
        return CookieSameSiteSupplier.ofLax().whenHasNameMatching("myapp.*");
    }

}
```

Kotlin

```
import org.springframework.boot.web.servlet.server.CookieSameSiteSupplier
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MySameSiteConfiguration {

    @Bean
    fun applicationCookieSameSiteSupplier(): CookieSameSiteSupplier {
        return CookieSameSiteSupplier.ofLax().whenHasNameMatching("myapp.*")
    }

}
```

Character Encoding

The character encoding behavior of the embedded servlet container for request and response handling can be configured using the `server.servlet.encoding.*` configuration properties.

When a request's `Accept-Language` header indicates a locale for the request it will be automatically mapped to a charset by the servlet container. Each container provides default locale to charset mappings and you should verify that they meet your application's needs. When they do not, use the `server.servlet.encoding.mapping` configuration property to customize the mappings, as shown in the following example:

Properties

```
server.servlet.encoding.mapping.ko=UTF-8
```

Yaml

```
server:
  servlet:
    encoding:
      mapping:
        ko: "UTF-8"
```

In the preceding example, the `ko` (Korean) locale has been mapped to `UTF-8`. This is equivalent to a `<locale-encoding-mapping-list>` entry in a `web.xml` file of a traditional war deployment.

Programmatic Customization

If you need to programmatically configure your embedded servlet container, you can register a Spring bean that implements the `WebServerFactoryCustomizer` interface. `WebServerFactoryCustomizer` provides access to the `ConfigurableServletWebServerFactory`, which includes numerous customization setter methods. The following example shows programmatically setting the port:

Java

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.stereotype.Component;

@Component
public class MyWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory server) {
        server.setPort(9000);
    }

}
```

Kotlin

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory
import org.springframework.stereotype.Component

@Component
class MyWebServerFactoryCustomizer :
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    override fun customize(server: ConfigurableServletWebServerFactory) {
        server.setPort(9000)
    }

}
```

[TomcatServletWebServerFactory](#), [JettyServletWebServerFactory](#) and [UndertowServletWebServerFactory](#) are dedicated variants of [ConfigurableServletWebServerFactory](#) that have additional customization setter methods for Tomcat, Jetty and Undertow respectively. The following example shows how to customize [TomcatServletWebServerFactory](#) that provides access to Tomcat-specific configuration options:

Java

```
import java.time.Duration;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyTomcatWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    @Override
    public void customize(TomcatServletWebServerFactory server) {
        server.addConnectorCustomizers((connector) ->
connector.setAsyncTimeout(Duration.ofSeconds(20).toMillis()));
    }

}
```

Kotlin

```
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.stereotype.Component
import java.time.Duration

@Component
class MyTomcatWebServerFactoryCustomizer :
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    override fun customize(server: TomcatServletWebServerFactory) {
        server.addConnectorCustomizers({ connector -> connector.asyncTimeout =
Duration.ofSeconds(20).toMillis() })
    }

}
```

Customizing ConfigurableServletWebServerFactory Directly

For more advanced use cases that require you to extend from [ServletWebServerFactory](#), you can expose a bean of such type yourself.

Setters are provided for many configuration options. Several protected method “hooks” are also provided should you need to do something more exotic. See the [source code documentation](#) for details.

NOTE

Auto-configured customizers are still applied on your custom factory, so use that option carefully.

JSP Limitations

When running a Spring Boot application that uses an embedded servlet container (and is packaged as an executable archive), there are some limitations in the JSP support.

- With Jetty and Tomcat, it should work if you use war packaging. An executable war will work when launched with `java -jar`, and will also be deployable to any standard container. JSPs are not supported when using an executable jar.
- Undertow does not support JSPs.
- Creating a custom `error.jsp` page does not override the default view for [error handling](#). [Custom error pages](#) should be used instead.

8.2. Reactive Web Applications

Spring Boot simplifies development of reactive web applications by providing auto-configuration for Spring WebFlux.

8.2.1. The “Spring WebFlux Framework”

Spring WebFlux is the new reactive web framework introduced in Spring Framework 5.0. Unlike Spring MVC, it does not require the servlet API, is fully asynchronous and non-blocking, and implements the [Reactive Streams](#) specification through [the Reactor project](#).

Spring WebFlux comes in two flavors: functional and annotation-based. The annotation-based one is quite close to the Spring MVC model, as shown in the following example:

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class MyRestController {

    private final UserRepository userRepository;

    private final CustomerRepository customerRepository;

    public MyRestController(UserRepository userRepository, CustomerRepository
customerRepository) {
        this.userRepository = userRepository;
        this.customerRepository = customerRepository;
    }

    @GetMapping("/{userId}")
    public Mono<User> getUser(@PathVariable Long userId) {
        return this.userRepository.findById(userId);
    }

    @GetMapping("/{userId}/customers")
    public Flux<Customer> getUserCustomers(@PathVariable Long userId) {
        return
this.userRepository.findById(userId).flatMapMany(this.customerRepository::findByUser);
    }

    @DeleteMapping("/{userId}")
    public Mono<Void> deleteUser(@PathVariable Long userId) {
        return this.userRepository.deleteById(userId);
    }

}
```

```
import org.springframework.web.bind.annotation.DeleteMapping
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import reactor.core.publisher.Flux
import reactor.core.publisher.Mono

@RestController
@RequestMapping("/users")
class MyRestController(private val userRepository: UserRepository, private val
customerRepository: CustomerRepository) {

    @GetMapping("/{userId}")
    fun getUser(@PathVariable userId: Long): Mono<User?> {
        return userRepository.findById(userId)
    }

    @GetMapping("/{userId}/customers")
    fun getUserCustomers(@PathVariable userId: Long): Flux<Customer> {
        return userRepository.findById(userId).flatMapMany { user: User? ->
            customerRepository.findByUser(user)
        }
    }

    @DeleteMapping("/{userId}")
    fun deleteUser(@PathVariable userId: Long): Mono<Void> {
        return userRepository.deleteById(userId)
    }

}
```

WebFlux is part of the Spring Framework and detailed information is available in its [reference documentation](#).

“WebFlux.fn”, the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import org.springframework.web.reactive.function.server.RequestPredicate;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;

import static
org.springframework.web.reactive.function.server.RequestPredicates.accept;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;

@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {

    private static final RequestPredicate ACCEPT_JSON =
accept(MediaType.APPLICATION_JSON);

    @Bean
    public RouterFunction<ServerResponse> monoRouterFunction(MyUserHandler
userHandler) {
        return route()
            .GET("/{user}", ACCEPT_JSON, userHandler::getUser)
            .GET("/{user}/customers", ACCEPT_JSON, userHandler::getUserCustomers)
            .DELETE("/{user}", ACCEPT_JSON, userHandler::deleteUser)
            .build();
    }
}
```

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.MediaType
import org.springframework.web.reactive.function.server.RequestPredicates.DELETE
import org.springframework.web.reactive.function.server.RequestPredicates.GET
import org.springframework.web.reactive.function.server.RequestPredicates.accept
import org.springframework.web.reactive.function.server.RouterFunction
import org.springframework.web.reactive.function.server.RouterFunctions
import org.springframework.web.reactive.function.server.ServerResponse

@Configuration(proxyBeanMethods = false)
class MyRoutingConfiguration {

    @Bean
    fun monoRouterFunction(userHandler: MyUserHandler): RouterFunction<ServerResponse>
    {
        return RouterFunctions.route(
            GET("/{user}").and(ACCEPT_JSON), userHandler::getUser).andRoute(
            GET("/{user}/customers").and(ACCEPT_JSON),
            userHandler::getUserCustomers).andRoute(
                DELETE("/{user}").and(ACCEPT_JSON), userHandler::deleteUser)
    }

    companion object {
        private val ACCEPT_JSON = accept(MediaType.APPLICATION_JSON)
    }
}
```

Java

```
import reactor.core.publisher.Mono;

import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;

@Component
public class MyUserHandler {

    public Mono<ServerResponse> getUser(ServerRequest request) {
        ...
    }

    public Mono<ServerResponse> getUserCustomers(ServerRequest request) {
        ...
    }

    public Mono<ServerResponse> deleteUser(ServerRequest request) {
        ...
    }

}
```

Kotlin

```
import org.springframework.stereotype.Component
import org.springframework.web.reactive.function.server.ServerRequest
import org.springframework.web.reactive.function.server.ServerResponse
import reactor.core.publisher.Mono

@Component
class MyUserHandler {

    fun getUser(request: ServerRequest?): Mono<ServerResponse> {
        return ServerResponse.ok().build()
    }

    fun getUserCustomers(request: ServerRequest?): Mono<ServerResponse> {
        return ServerResponse.ok().build()
    }

    fun deleteUser(request: ServerRequest?): Mono<ServerResponse> {
        return ServerResponse.ok().build()
    }

}
```

“WebFlux.fn” is part of the Spring Framework and detailed information is available in its [reference](#)

documentation.

TIP You can define as many `RouterFunction` beans as you like to modularize the definition of the router. Beans can be ordered if you need to apply a precedence.

To get started, add the `spring-boot-starter-webflux` module to your application.

NOTE Adding both `spring-boot-starter-web` and `spring-boot-starter-webflux` modules in your application results in Spring Boot auto-configuring Spring MVC, not WebFlux. This behavior has been chosen because many Spring developers add `spring-boot-starter-webflux` to their Spring MVC application to use the reactive `WebClient`. You can still enforce your choice by setting the chosen application type to `SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`.

Spring WebFlux Auto-configuration

Spring Boot provides auto-configuration for Spring WebFlux that works well with most applications.

The auto-configuration adds the following features on top of Spring's defaults:

- Configuring codecs for `HttpMessageReader` and `HttpMessageWriter` instances (described [later in this document](#)).
- Support for serving static resources, including support for WebJars (described [later in this document](#)).

If you want to keep Spring Boot WebFlux features and you want to add additional `WebFlux configuration`, you can add your own `@Configuration` class of type `WebFluxConfigurer` but **without** `@EnableWebFlux`.

If you want to take complete control of Spring WebFlux, you can add your own `@Configuration` annotated with `@EnableWebFlux`.

Spring WebFlux Conversion Service

If you want to customize the `ConversionService` used by Spring WebFlux, you can provide a `WebFluxConfigurer` bean with an `addFormatters` method.

Conversion can also be customized using the `spring.webflux.format.*` configuration properties. When not configured, the following defaults are used:

Property	DateFormatter
<code>spring.webflux.format.date</code>	<code>ofLocalizedDate(FormatStyle.SHORT)</code>
<code>spring.webflux.format.time</code>	<code>ofLocalizedTime(FormatStyle.SHORT)</code>
<code>spring.webflux.format.date-time</code>	<code>ofLocalDateTime(FormatStyle.SHORT)</code>

HTTP Codecs with `HttpMessageReaders` and `HttpMessageWriters`

Spring WebFlux uses the `HttpMessageReader` and `HttpMessageWriter` interfaces to convert HTTP requests and responses. They are configured with `CodecConfigurer` to have sensible defaults by looking at the libraries available in your classpath.

Spring Boot provides dedicated configuration properties for codecs, `spring.codec.*`. It also applies further customization by using `CodecCustomizer` instances. For example, `spring.jackson.*` configuration keys are applied to the Jackson codec.

If you need to add or customize codecs, you can create a custom `CodecCustomizer` component, as shown in the following example:

Java

```
import org.springframework.boot.web.codec.CodecCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.codec.ServerSentEventHttpMessageReader;

@Configuration(proxyBeanMethods = false)
public class MyCodecsConfiguration {

    @Bean
    public CodecCustomizer myCodecCustomizer() {
        return (configurer) -> {
            configurer.registerDefaults(false);
            configurer.customCodecs().register(new
ServerSentEventHttpMessageReader());
            // ...
        };
    }
}
```

```
import org.springframework.boot.web.codec.CodecCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.http.codec.CodecConfigurer
import org.springframework.http.codec.ServerSentEventHttpMessageReader

class MyCodecsConfiguration {

    @Bean
    fun myCodecCustomizer(): CodecCustomizer {
        return CodecCustomizer { configurer: CodecConfigurer ->
            configurer.registerDefaults(false)
            configurer.customCodecs().register(ServerSentEventHttpMessageReader())
        }
    }

}
```

You can also leverage [Boot's custom JSON serializers and deserializers](#).

Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath. It uses the `ResourceWebHandler` from Spring WebFlux so that you can modify that behavior by adding your own `WebFluxConfigurer` and overriding the `addResourceHandlers` method.

By default, resources are mapped on `/**`, but you can tune that by setting the `spring.webflux.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

Properties

```
spring.webflux.static-path-pattern=/resources/**
```

Yaml

```
spring:
  webflux:
    static-path-pattern: "/resources/**"
```

You can also customize the static resource locations by using `spring.web.resources.static-locations`. Doing so replaces the default values with a list of directory locations. If you do so, the default welcome page detection switches to your custom locations. So, if there is an `index.html` in any of your locations on startup, it is the home page of the application.

In addition to the “standard” static resource locations listed earlier, a special case is made for [Webjars content](#). By default, any resources with a path in `/webjars/**` are served from jar files if

they are packaged in the Webjars format. The path can be customized with the `spring.webflux.webjars-path-pattern` property.

TIP Spring WebFlux applications do not strictly depend on the servlet API, so they cannot be deployed as war files and do not use the `src/main/webapp` directory.

Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

This only acts as a fallback for actual index routes defined by the application. The ordering is defined by the order of `HandlerMapping` beans which is by default the following:

<code>RouterFunctionMapping</code>	Endpoints declared with <code>RouterFunction</code> beans
<code>RequestMappingHandlerMapping</code>	Endpoints declared in <code>@Controller</code> beans
<code>RouterFunctionMapping</code> for the Welcome Page	The welcome page support

Template Engines

As well as REST web services, you can also use Spring WebFlux to serve dynamic HTML content. Spring WebFlux supports a variety of templating technologies, including Thymeleaf, FreeMarker, and Mustache.

Spring Boot includes auto-configuration support for the following templating engines:

- [FreeMarker](#)
- [Thymeleaf](#)
- [Mustache](#)

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

Error Handling

Spring Boot provides a `WebExceptionHandler` that handles all errors in a sensible way. Its position in the processing order is immediately before the handlers provided by WebFlux, which are considered last. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a “whitelabel” error handler that renders the same data in HTML format. You can also provide your own HTML templates to display errors (see the [next section](#)).

Before customizing error handling in Spring Boot directly, you can leverage the [RFC 7807 Problem Details](#) support in Spring WebFlux. Spring WebFlux can produce custom error messages with the `application/problem+json` media type, like:

```
{  
  "type": "https://example.org/problems/unknown-project",  
  "title": "Unknown project",  
  "status": 404,  
  "detail": "No project found for id 'spring-unknown'",  
  "instance": "/projects/spring-unknown"  
}
```

This support can be enabled by setting `spring.webflux.problemdetails.enabled` to `true`.

The first step to customizing this feature often involves using the existing mechanism but replacing or augmenting the error contents. For that, you can add a bean of type `ErrorAttributes`.

To change the error handling behavior, you can implement `ErrorWebExceptionHandler` and register a bean definition of that type. Because an `ErrorWebExceptionHandler` is quite low-level, Spring Boot also provides a convenient `AbstractErrorWebExceptionHandler` to let you handle errors in a WebFlux functional way, as shown in the following example:

Java

```
import reactor.core.publisher.Mono;

import org.springframework.boot.autoconfigure.web.WebProperties;
import
org.springframework.boot.autoconfigure.web.reactive.error.AbstractErrorWebExceptionHandler;
import org.springframework.boot.web.reactive.error.ErrorAttributes;
import org.springframework.context.ApplicationContext;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.codec.ServerCodecConfigurer;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import org.springframework.web.reactive.function.server.ServerResponse.BodyBuilder;

@Component
public class MyErrorWebExceptionHandler extends AbstractErrorWebExceptionHandler {

    public MyErrorWebExceptionHandler(ErrorAttributes errorAttributes, WebProperties webProperties,
                                      ApplicationContext applicationContext, ServerCodecConfigurer serverCodecConfigurer) {
        super(errorAttributes, webProperties.getResources(), applicationContext);
        setMessageReaders(serverCodecConfigurer.getReaders());
        setMessageWriters(serverCodecConfigurer.getWriters());
    }

    @Override
    protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes errorAttributes) {
        return RouterFunctions.route(this::acceptsXml, this::handleErrorAsXml);
    }

    private boolean acceptsXml(ServerRequest request) {
        return request.headers().accept().contains(MediaType.APPLICATION_XML);
    }

    public Mono<ServerResponse> handleErrorAsXml(ServerRequest request) {
        BodyBuilder builder = ServerResponse.status(HttpStatus.INTERNAL_SERVER_ERROR);
        // ... additional builder calls
        return builder.build();
    }

}
```

```

import org.springframework.boot.autoconfigure.web.WebProperties
import
org.springframework.boot.autoconfigure.web.reactive.error.AbstractErrorWebExceptionHandler
import org.springframework.boot.web.reactive.error.ErrorAttributes
import org.springframework.context.ApplicationContext
import org.springframework.http.HttpStatus
import org.springframework.http.MediaType
import org.springframework.http.codec.ServerCodecConfigurer
import org.springframework.stereotype.Component
import org.springframework.web.reactive.function.server.RouterFunction
import org.springframework.web.reactive.function.server.RouterFunctions
import org.springframework.web.reactive.function.server.ServerRequest
import org.springframework.web.reactive.function.server.ServerResponse
import reactor.core.publisher.Mono

@Component
class MyErrorWebExceptionHandler(
    errorAttributes: ErrorAttributes, webProperties: WebProperties,
    applicationContext: ApplicationContext, serverCodecConfigurer:
ServerCodecConfigurer
) : AbstractErrorWebExceptionHandler(errorAttributes, webProperties.resources,
applicationContext) {

    init {
        setMessageReaders(serverCodecConfigurer.readers)
        setMessageWriters(serverCodecConfigurer.writers)
    }

    override fun getRoutingFunction(errorAttributes: ErrorAttributes):
RouterFunction<ServerResponse> {
        return RouterFunctions.route(this::acceptsXml, this::handleErrorAsXml)
    }

    private fun acceptsXml(request: ServerRequest): Boolean {
        return request.headers().accept().contains(MediaType.APPLICATION_XML)
    }

    fun handleErrorAsXml(request: ServerRequest): Mono<ServerResponse> {
        val builder = ServerResponse.status(HttpStatus.INTERNAL_SERVER_ERROR)
        // ... additional builder calls
        return builder.build()
    }

}

```

For a more complete picture, you can also subclass [DefaultErrorWebExceptionHandler](#) directly and override specific methods.

In some cases, errors handled at the controller level are not recorded by web observations or the [metrics infrastructure](#). Applications can ensure that such exceptions are recorded with the observations by [setting the handled exception on the observation context](#).

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add views that resolve from `error/*`, for example by adding files to a `/error` directory. Error pages can either be static HTML (that is, added under any of the static resource directories) or built with templates. The name of the file should be the exact status code, a status code series mask, or `error` for a default if nothing else matches. Note that the path to the default error view is `error/error`, whereas with Spring MVC the default error view is `error`.

For example, to map `404` to a static HTML file, your directory structure would be as follows:

```
src/
+- main/
  +- java/
  |   + <source code>
  +- resources/
    +- public/
      +- error/
        |   +- 404.html
        +- <other public assets>
```

To map all `5xx` errors by using a Mustache template, your directory structure would be as follows:

```
src/
+- main/
  +- java/
  |   + <source code>
  +- resources/
    +- templates/
      +- error/
        |   +- 5xx.mustache
        +- <other templates>
```

Web Filters

Spring WebFlux provides a `WebFilter` interface that can be implemented to filter HTTP request-response exchanges. `WebFilter` beans found in the application context will be automatically used to filter each exchange.

Where the order of the filters is important they can implement `Ordered` or be annotated with `@Order`. Spring Boot auto-configuration may configure web filters for you. When it does so, the orders shown in the following table will be used:

Web Filter	Order
<code>WebFilterChainProxy</code> (Spring Security)	-100
<code>HttpExchangesWebFilter</code>	<code>Ordered.LOWEST_PRECEDENCE - 10</code>

8.2.2. Embedded Reactive Server Support

Spring Boot includes support for the following embedded reactive web servers: Reactor Netty, Tomcat, Jetty, and Undertow. Most developers use the appropriate “Starter” to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port 8080.

Customizing Reactive Servers

Common reactive web server settings can be configured by using Spring `Environment` properties. Usually, you would define the properties in your `application.properties` or `application.yaml` file.

Common server settings include:

- Network settings: Listen port for incoming HTTP requests (`server.port`), interface address to bind to (`server.address`), and so on.
- Error management: Location of the error page (`server.error.path`) and so on.
- [SSL](#)
- [HTTP compression](#)

Spring Boot tries as much as possible to expose common settings, but this is not always possible. For those cases, dedicated namespaces such as `server.netty.*` offer server-specific customizations.

TIP See the `ServerProperties` class for a complete list.

Programmatic Customization

If you need to programmatically configure your reactive web server, you can register a Spring bean that implements the `WebServerFactoryCustomizer` interface. `WebServerFactoryCustomizer` provides access to the `ConfigurableReactiveWebServerFactory`, which includes numerous customization setter methods. The following example shows programmatically setting the port:

Java

```
import org.springframework.boot.web.reactive.server.ConfigurableReactiveWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<ConfigurableReactiveWebServerFactory> {

    @Override
    public void customize(ConfigurableReactiveWebServerFactory server) {
        server.setPort(9000);
    }

}
```

Kotlin

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.boot.web.reactive.server.ConfigurableReactiveWebServerFactory
import org.springframework.stereotype.Component

@Component
class MyWebServerFactoryCustomizer :
WebServerFactoryCustomizer<ConfigurableReactiveWebServerFactory> {

    override fun customize(server: ConfigurableReactiveWebServerFactory) {
        server.setPort(9000)
    }

}
```

`JettyReactiveWebServerFactory`, `NettyReactiveWebServerFactory`, `TomcatReactiveWebServerFactory`, and `UndertowReactiveWebServerFactory` are dedicated variants of `ConfigurableReactiveWebServerFactory` that have additional customization setter methods for Jetty, Reactor Netty, Tomcat, and Undertow respectively. The following example shows how to customize `NettyReactiveWebServerFactory` that provides access to Reactor Netty-specific configuration options:

Java

```
import java.time.Duration;

import org.springframework.boot.web.embedded.netty.NettyReactiveWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.stereotype.Component;

@Component
public class MyNettyWebServerFactoryCustomizer implements
WebServerFactoryCustomizer<NettyReactiveWebServerFactory> {

    @Override
    public void customize(NettyReactiveWebServerFactory factory) {
        factory.addServerCustomizers((server) ->
server.idleTimeout(Duration.ofSeconds(20)));
    }

}
```

Kotlin

```
import org.springframework.boot.web.embedded.netty.NettyReactiveWebServerFactory
import org.springframework.boot.web.server.WebServerFactoryCustomizer
import org.springframework.stereotype.Component
import java.time.Duration

@Component
class MyNettyWebServerFactoryCustomizer :
WebServerFactoryCustomizer<NettyReactiveWebServerFactory> {

    override fun customize(factory: NettyReactiveWebServerFactory) {
        factory.addServerCustomizers({ server ->
server.idleTimeout(Duration.ofSeconds(20)) })
    }

}
```

Customizing ConfigurableReactiveWebServerFactory Directly

For more advanced use cases that require you to extend from [ReactiveWebServerFactory](#), you can expose a bean of such type yourself.

Setters are provided for many configuration options. Several protected method “hooks” are also provided should you need to do something more exotic. See the [source code documentation](#) for details.

NOTE

Auto-configured customizers are still applied on your custom factory, so use that option carefully.

8.2.3. Reactive Server Resources Configuration

When auto-configuring a Reactor Netty or Jetty server, Spring Boot will create specific beans that will provide HTTP resources to the server instance: `ReactorResourceFactory` or `JettyResourceFactory`.

By default, those resources will be also shared with the Reactor Netty and Jetty clients for optimal performances, given:

- the same technology is used for server and client
- the client instance is built using the `WebClient.Builder` bean auto-configured by Spring Boot

Developers can override the resource configuration for Jetty and Reactor Netty by providing a custom `ReactorResourceFactory` or `JettyResourceFactory` bean - this will be applied to both clients and servers.

You can learn more about the resource configuration on the client side in the [WebClient Runtime section](#).

8.3. Graceful Shutdown

Graceful shutdown is supported with all four embedded web servers (Jetty, Reactor Netty, Tomcat, and Undertow) and with both reactive and servlet-based web applications. It occurs as part of closing the application context and is performed in the earliest phase of stopping `SmartLifecycle` beans. This stop processing uses a timeout which provides a grace period during which existing requests will be allowed to complete but no new requests will be permitted.

The exact way in which new requests are not permitted varies depending on the web server that is being used. Implementations may stop accepting requests at the network layer, or they may return a response with a specific HTTP status code or HTTP header. The use of persistent connections can also change the way that requests stop being accepted.

TIP To learn about more the specific method used with your web server, see the `shutDownGracefully` javadoc for `TomcatWebServer`, `NettyWebServer`, `JettyWebServer` or `UndertowWebServer`.

Jetty, Reactor Netty, and Tomcat will stop accepting new requests at the network layer. Undertow will accept new connections but respond immediately with a service unavailable (503) response.

NOTE Graceful shutdown with Tomcat requires Tomcat 9.0.33 or later.

To enable graceful shutdown, configure the `server.shutdown` property, as shown in the following example:

Properties

```
server.shutdown=graceful
```

Yaml

```
server:  
  shutdown: "graceful"
```

To configure the timeout period, configure the `spring.lifecycle.timeout-per-shutdown-phase` property, as shown in the following example:

Properties

```
spring.lifecycle.timeout-per-shutdown-phase=20s
```

Yaml

```
spring:  
  lifecycle:  
    timeout-per-shutdown-phase: "20s"
```

IMPORTANT

Using graceful shutdown with your IDE may not work properly if it does not send a proper `SIGTERM` signal. See the documentation of your IDE for more details.

8.4. Spring Security

If [Spring Security](#) is on the classpath, then web applications are secured by default. Spring Boot relies on Spring Security's content-negotiation strategy to determine whether to use `httpBasic` or `formLogin`. To add method-level security to a web application, you can also add `@EnableGlobalMethodSecurity` with your desired settings. Additional information can be found in the [Spring Security Reference Guide](#).

The default `UserDetailsService` has a single user. The user name is `user`, and the password is random and is printed at `WARN` level when the application starts, as shown in the following example:

```
Using generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35
```

This generated password is for development use only. Your security configuration must be updated before running your application in production.

NOTE

If you fine-tune your logging configuration, ensure that the `org.springframework.boot.autoconfigure.security` category is set to log `WARN`-level messages. Otherwise, the default password is not printed.

You can change the username and password by providing a `spring.security.user.name` and `spring.security.user.password`.

The basic features you get by default in a web application are:

- A `UserDetailsService` (or `ReactiveUserDetailsService` in case of a WebFlux application) bean with in-memory store and a single user with a generated password (see `SecurityProperties.User` for the properties of the user).
- Form-based login or HTTP Basic security (depending on the `Accept` header in the request) for the entire application (including actuator endpoints if actuator is on the classpath).
- A `DefaultAuthenticationEventPublisher` for publishing authentication events.

You can provide a different `AuthenticationEventPublisher` by adding a bean for it.

8.4.1. MVC Security

The default security configuration is implemented in `SecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `SecurityAutoConfiguration` imports `SpringBootWebSecurityConfiguration` for web security and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications.

To switch off the default web application security configuration completely or to combine multiple Spring Security components such as OAuth2 Client and Resource Server, add a bean of type `SecurityFilterChain` (doing so does not disable the `UserDetailsService` configuration or Actuator's security). To also switch off the `UserDetailsService` configuration, you can add a bean of type `UserDetailsService`, `AuthenticationProvider`, or `AuthenticationManager`.

The auto-configuration of a `UserDetailsService` will also back off any of the following Spring Security modules is on the classpath:

- `spring-security-oauth2-client`
- `spring-security-oauth2-resource-server`
- `spring-security-saml2-service-provider`

To use `UserDetailsService` in addition to one or more of these dependencies, define your own `InMemoryUserDetailsManager` bean.

Access rules can be overridden by adding a custom `SecurityFilterChain` bean. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `RequestMatcher` that is based on the `management.endpoints.web.base-path` property. `PathRequest` can be used to create a `RequestMatcher` for resources in commonly used locations.

8.4.2. WebFlux Security

Similar to Spring MVC applications, you can secure your WebFlux applications by adding the `spring-boot-starter-security` dependency. The default security configuration is implemented in `ReactiveSecurityAutoConfiguration` and `UserDetailsServiceAutoConfiguration`. `ReactiveSecurityAutoConfiguration` imports `WebFluxSecurityConfiguration` for web security and `UserDetailsServiceAutoConfiguration` configures authentication, which is also relevant in non-web applications.

To switch off the default web application security configuration completely, you can add a bean of type `WebFilterChainProxy` (doing so does not disable the `UserDetailsService` configuration or Actuator's security). To also switch off the `UserDetailsService` configuration, you can add a bean of type `ReactiveUserDetailsService` or `ReactiveAuthenticationManager`.

The auto-configuration will also back off when any of the following Spring Security modules is on the classpath:

- `spring-security-oauth2-client`
- `spring-security-oauth2-resource-server`

To use `ReactiveUserDetailsService` in addition to one or more of these dependencies, define your own `MapReactiveUserDetailsService` bean.

Access rules and the use of multiple Spring Security components such as OAuth 2 Client and Resource Server can be configured by adding a custom `SecurityWebFilterChain` bean. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `ServerWebExchangeMatcher` that is based on the `management.endpoints.web.base-path` property.

`PathRequest` can be used to create a `ServerWebExchangeMatcher` for resources in commonly used locations.

For example, you can customize your security configuration by adding something like:

Java

```
import org.springframework.boot.autoconfigure.security.reactive.PathRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration(proxyBeanMethods = false)
public class MyWebFluxSecurityConfiguration {

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http.authorizeExchange((exchange) -> {

            exchange.matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll();
            exchange.pathMatchers("/foo", "/bar").authenticated();
        });
        http.formLogin(withDefaults());
        return http.build();
    }

}
```

```

import org.springframework.boot.autoconfigure.security.reactive.PathRequest
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.config.Customizer.withDefaults
import org.springframework.security.config.web.server.ServerHttpSecurity
import org.springframework.security.web.server.SecurityWebFilterChain

@Configuration(proxyBeanMethods = false)
class MyWebFluxSecurityConfiguration {

    @Bean
    fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
        http.authorizeExchange { spec ->

            spec.matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
                spec.pathMatchers("/foo", "/bar").authenticated()
        }
        http.formLogin(withDefaults())
        return http.build()
    }

}

```

8.4.3. OAuth2

[OAuth2](#) is a widely used authorization framework that is supported by Spring.

Client

If you have [spring-security-oauth2-client](#) on your classpath, you can take advantage of some auto-configuration to set up OAuth2/Open ID Connect clients. This configuration makes use of the properties under [OAuth2ClientProperties](#). The same properties are applicable to both servlet and reactive applications.

You can register multiple OAuth2 clients and providers under the [spring.security.oauth2.client](#) prefix, as shown in the following example:

Properties

```

spring.security.oauth2.client.registration.my-login-client.client-id=abcd
spring.security.oauth2.client.registration.my-login-client.client-secret=password
spring.security.oauth2.client.registration.my-login-client.client-name=Client for
OpenID Connect
spring.security.oauth2.client.registration.my-login-client.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-login-
client.scope=openid,profile,email,phone,address
spring.security.oauth2.client.registration.my-login-client.redirect-
uri={baseUrl}/login/oauth2/code/{registrationId}

```

```
spring.security.oauth2.client.registration.my-login-client.client-authentication-
method=client_secret_basic
spring.security.oauth2.client.registration.my-login-client.authorization-grant-
type=authorization_code

spring.security.oauth2.client.registration.my-client-1.client-id=abcd
spring.security.oauth2.client.registration.my-client-1.client-secret=password
spring.security.oauth2.client.registration.my-client-1.client-name=Client for user
scope
spring.security.oauth2.client.registration.my-client-1.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-1.scope=user
spring.security.oauth2.client.registration.my-client-1.redirect-
uri={baseUrl}/authorized/user
spring.security.oauth2.client.registration.my-client-1.client-authentication-
method=client_secret_basic
spring.security.oauth2.client.registration.my-client-1.authorization-grant-
type=authorization_code

spring.security.oauth2.client.registration.my-client-2.client-id=abcd
spring.security.oauth2.client.registration.my-client-2.client-secret=password
spring.security.oauth2.client.registration.my-client-2.client-name=Client for email
scope
spring.security.oauth2.client.registration.my-client-2.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-2.scope=email
spring.security.oauth2.client.registration.my-client-2.redirect-
uri={baseUrl}/authorized/email
spring.security.oauth2.client.registration.my-client-2.client-authentication-
method=client_secret_basic
spring.security.oauth2.client.registration.my-client-2.authorization-grant-
type=authorization_code

spring.security.oauth2.client.provider.my-oauth-provider.authorization-uri=https://my-
auth-server.com/oauth2/authorize
spring.security.oauth2.client.provider.my-oauth-provider.token-uri=https://my-auth-
server.com/oauth2/token
spring.security.oauth2.client.provider.my-oauth-provider.user-info-uri=https://my-
auth-server.com/userinfo
spring.security.oauth2.client.provider.my-oauth-provider.user-info-authentication-
method=header
spring.security.oauth2.client.provider.my-oauth-provider.jwk-set-uri=https://my-auth-
server.com/oauth2/jwks
spring.security.oauth2.client.provider.my-oauth-provider.user-name-attribute=name
```

```

spring:
  security:
    oauth2:
      client:
        registration:
          my-login-client:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for OpenID Connect"
            provider: "my-oauth-provider"
            scope: "openid,profile,email,phone,address"
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            client-authentication-method: "client_secret_basic"
            authorization-grant-type: "authorization_code"

          my-client-1:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for user scope"
            provider: "my-oauth-provider"
            scope: "user"
            redirect-uri: "{baseUrl}/authorized/user"
            client-authentication-method: "client_secret_basic"
            authorization-grant-type: "authorization_code"

          my-client-2:
            client-id: "abcd"
            client-secret: "password"
            client-name: "Client for email scope"
            provider: "my-oauth-provider"
            scope: "email"
            redirect-uri: "{baseUrl}/authorized/email"
            client-authentication-method: "client_secret_basic"
            authorization-grant-type: "authorization_code"

      provider:
        my-oauth-provider:
          authorization-uri: "https://my-auth-server.com/oauth2/authorize"
          token-uri: "https://my-auth-server.com/oauth2/token"
          user-info-uri: "https://my-auth-server.com/userinfo"
          user-info-authentication-method: "header"
          jwk-set-uri: "https://my-auth-server.com/oauth2/jwks"
          user-name-attribute: "name"

```

For OpenID Connect providers that support [OpenID Connect discovery](#), the configuration can be further simplified. The provider needs to be configured with an **issuer-uri** which is the URI that it asserts as its Issuer Identifier. For example, if the **issuer-uri** provided is "https://example.com", then an "OpenID Provider Configuration Request" will be made to "https://example.com/.well-

known/openid-configuration". The result is expected to be an "OpenID Provider Configuration Response". The following example shows how an OpenID Connect Provider can be configured with the `issuer-uri`:

Properties

```
spring.security.oauth2.client.provider.oidc-provider.issuer-uri=https://dev-123456.oktapreview.com/oauth2/default/
```

Yaml

```
spring:
  security:
    oauth2:
      client:
        provider:
          oidc-provider:
            issuer-uri: "https://dev-123456.oktapreview.com/oauth2/default/"
```

By default, Spring Security's `OAuth2LoginAuthenticationFilter` only processes URLs matching `/login/oauth2/code/*`. If you want to customize the `redirect-uri` to use a different pattern, you need to provide configuration to process that custom pattern. For example, for servlet applications, you can add your own `SecurityFilterChain` that resembles the following:

Java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration(proxyBeanMethods = false)
@EnableWebSecurity
public class MyOAuthClientConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
    {
        http
            .authorizeHttpRequests((requests) -> requests
                .anyRequest().authenticated()
            )
            .oauth2Login((login) -> login
                . redirectionEndpoint((endpoint) -> endpoint
                    .baseUri("/login/oauth2/callback/*")
                )
            );
        return http.build();
    }

}
```

```

import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
import org.springframework.security.config.annotation.web.invoke
import org.springframework.security.web.SecurityFilterChain

@Configuration(proxyBeanMethods = false)
@EnableWebSecurity
open class MyOAuthClientConfiguration {

    @Bean
    open fun securityFilterChain(http: HttpSecurity): SecurityFilterChain {
        http {
            authorizeHttpRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2Login {
                redirectionEndpoint {
                    baseUri = "/login/oauth2/callback/*"
                }
            }
        }
        return http.build()
    }
}

```

TIP Spring Boot auto-configures an `InMemoryOAuth2AuthorizedClientService` which is used by Spring Security for the management of client registrations. The `InMemoryOAuth2AuthorizedClientService` has limited capabilities and we recommend using it only for development environments. For production environments, consider using a `JdbcOAuth2AuthorizedClientService` or creating your own implementation of `OAuth2AuthorizedClientService`.

OAuth2 Client Registration for Common Providers

For common OAuth2 and OpenID providers, including Google, Github, Facebook, and Okta, we provide a set of provider defaults (`google`, `github`, `facebook`, and `okta`, respectively).

If you do not need to customize these providers, you can set the `provider` attribute to the one for which you need to infer defaults. Also, if the key for the client registration matches a default supported provider, Spring Boot infers that as well.

In other words, the two configurations in the following example use the Google provider: