Spring Boot Reference Guide

1.2.0.RELEASE

Phillip Webb , Dave Syer , Josh Long , Stéphane Nicoll , Rob Winch ,
Andy Wilkinson , Marcel Overdijk , Christian Dupuis , Sébastien Deleuze

Table of Contents

I. Spring Boot Documentation 1
1. About the documentation 2
2. Getting help 3
3. First steps 4
4. Working with Spring Boot 5
5. Learning about Spring Boot features 6
6. Moving to production 7
7. Advanced topics 8
II. Getting started 9
8. Introducing Spring Boot 10
9. System Requirements 11
9.1. Servlet containers 11
10. Installing Spring Boot 12
10.1. Installation instructions for the Java developer 12
Maven installation 12
Gradle installation 13
10.2. Installing the Spring Boot CLI 14
Manual installation 14
Installation with GVM 14
OSX Homebrew installation 15
Command-line completion 15
Quick start Spring CLI example 15
10.3. Upgrading from an earlier version of Spring Boot 16
11. Developing your first Spring Boot application 17
11.1. Creating the POM 17
11.2. Adding classpath dependencies 18
11.3. Writing the code 18
The @RestController and @RequestMapping annotations 19
The @EnableAutoConfiguration annotation 19
The “main” method 19
11.4. Running the example 19
11.5. Creating an executable jar 20
12. What to read next 22
III. Using Spring Boot 23
13. Build systems 24
13.1. Maven 24
Inheriting the starter parent 24
Using Spring Boot without the parent POM 24
Changing the Java version 25
Using the Spring Boot Maven plugin 25
13.2. Gradle 25
13.3. Ant 26
13.4. Starter POMs 26
14. Structuring your code 29
14.1. Using the “default” package 29
14.2. Locating the main application class 29
15. Configuration classes 31

15.1. Importing additional configuration classes 31
15.2. Importing XML configuration 31
16. Auto-configuration 32
16.1. Gradually replacing auto-configuration 32
16.2. Disabling specific auto-configuration 32
17. Spring Beans and dependency injection 33
18. Using the @SpringBootApplication annotation 34
19. Running your application 35
19.1. Running from an IDE 35
19.2. Running as a packaged application 35
19.3. Using the Maven plugin 35
19.4. Using the Gradle plugin 36
19.5. Hot swapping 36
20. Packaging your application for production 37
21. What to read next 38
IV. Spring Boot features 39
22. SpringApplication 40
22.1. Customizing the Banner 40
22.2. Customizing SpringApplication 41
22.3. Fluent builder API 41
22.4. Application events and listeners 42
22.5. Web environment 42
22.6. Using the CommandLineRunner 42
22.7. Application exit 43
23. Externalized Configuration 44
23.1. Accessing command line properties 45
23.2. Application property files 45
23.3. Profile specific properties 46
23.4. Placeholders in properties 46
23.5. Using YAML instead of Properties 46
Loading YAML 46
Exposing YAML as properties in the Spring Environment 47
Multi-profile YAML documents 47
YAML shortcomings 48
23.6. Typesafe Configuration Properties 48
Relaxed binding 49
@ConfigurationProperties Validation 49
24. Profiles 50
24.1. Adding active profiles 50
24.2. Programmatically setting profiles 50
24.3. Profile specific configuration files 51
25. Logging 52
25.1. Log format 52
25.2. Console output 52
25.3. File output 53
25.4. Log Levels 53
25.5. Custom log configuration 53
26. Developing web applications 55
26.1. The ‘Spring Web MVC framework’ 55
Spring MVC auto-configuration 55

HttpMessageConverters 56
MessageCodesResolver 56
Static Content 56
Template engines 57
Error Handling 57
Error Handling on WebSphere Application Server 58
26.2. JAX-RS and Jersey 58
26.3. Embedded servlet container support 59
Servlets and Filters 59
The EmbeddedWebApplicationContext 59
Customizing embedded servlet containers 60
Programmatic customization 60
Customizing ConfigurableEmbeddedServletContainer directly 60
JSP limitations 60
27. Security 62
28. Working with SQL databases 64
28.1. Configure a DataSource 64
Embedded Database Support 64
Connection to a production database 64
Connection to a JNDI DataSource 65
28.2. Using JdbcTemplate 65
28.3. JPA and 'Spring Data' 66
Entity Classes 66
Spring Data JPA Repositories 67
Creating and dropping JPA databases 68
29. Working with NoSQL technologies 69
29.1. Redis 69
Connecting to Redis 69
29.2. MongoDB 69
Connecting to a MongoDB database 69
MongoTemplate 70
Spring Data MongoDB repositories 70
29.3. Gemfire 71
29.4. Solr 71
Connecting to Solr 71
Spring Data Solr repositories 72
29.5. Elasticsearch 72
Connecting to Elasticsearch 72
Spring Data Elasticsearch repositories 72
30. Messaging 73
30.1. JMS 73
HornetQ support 73
ActiveMQ support 74
Using a JNDI ConnectionFactory 74
Sending a message 74
Receiving a message 74
31. Sending email 76
32. Distributed Transactions with JTA 77
32.1. Using an Atomikos transaction manager 77
32.2. Using a Bitronix transaction manager 77

32.3. Using a Java EE managed transaction manager 78
32.4. Mixing XA and non-XA JMS connections 78
32.5. Supporting an alternative embedded transaction manager 78
33. Spring Integration 79
34. Monitoring and management over JMX 80
35. Testing 81
35.1. Test scope dependencies 81
35.2. Testing Spring applications 81
35.3. Testing Spring Boot applications 81
Using Spock to test Spring Boot applications 83
35.4. Test utilities 83
ConfigFileApplicationContextInitializer 83
EnvironmentTestUtils 83
OutputCapture 83
TestRestTemplate 84
36. Developing auto-configuration and using conditions 85
36.1. Understanding auto-configured beans 85
36.2. Locating auto-configuration candidates 85
36.3. Condition annotations 85
Class conditions 85
Bean conditions 85
Property conditions 86
Resource conditions 86
Web Application Conditions 86
SpEL expression conditions 86
37. WebSockets 87
38. What to read next 88
V. Spring Boot Actuator: Production-ready features 89
39. Enabling production-ready features. 90
40. Endpoints 91
40.1. Customizing endpoints 91
40.2. Health information 92
40.3. Security with HealthIndicators 92
Auto-configured HealthIndicators 92
Writing custom HealthIndicators 93
40.4. Custom application info information 93
Automatically expand info properties at build time 93
Automatic property expansion using Maven 94
Automatic property expansion using Gradle 94
Git commit information 95
41. Monitoring and management over HTTP 96
41.1. Securing sensitive endpoints 96
41.2. Customizing the management server context path 96
41.3. Customizing the management server port 96
41.4. Customizing the management server address 97
41.5. Disabling HTTP endpoints 97
41.6. HTTP Health endpoint access restrictions 97
42. Monitoring and management over JMX 98
42.1. Customizing MBean names 98
42.2. Disabling JMX endpoints 98

42.3. Using Jolokia for JMX over HTTP 98
Customizing Jolokia 98
Disabling Jolokia 98
43. Monitoring and management using a remote shell 99
43.1. Connecting to the remote shell 99
Remote shell credentials 99
43.2. Extending the remote shell 99
Remote shell commands 99
Remote shell plugins 100
44. Metrics 101
44.1. System metrics 101
44.2. DataSource metrics 102
44.3. Tomcat session metrics 102
44.4. Recording your own metrics 102
44.5. Adding your own public metrics 103
44.6. Metric repositories 103
44.7. Dropwizard Metrics 103
44.8. Message channel integration 104
45. Auditing 105
46. Tracing 106
46.1. Custom tracing 106
47. Process monitoring 107
47.1. Extend configuration 107
47.2. Programmatically 107
48. What to read next 108
VI. Deploying to the cloud 109
49. Cloud Foundry 110
49.1. Binding to services 110
50. Heroku 112
51. Openshift 114
52. Google App Engine 115
53. What to read next 116
VII. Spring Boot CLI 117
54. Installing the CLI 118
55. Using the CLI 119
55.1. Running applications using the CLI 119
Deduced “grab” dependencies 120
Deduced “grab” coordinates 121
Default import statements 121
Automatic main method 121
Custom “grab” metadata 121
55.2. Testing your code 122
55.3. Applications with multiple source files 122
55.4. Packaging your application 122
55.5. Initialize a new project 123
55.6. Using the embedded shell 123
55.7. Adding extensions to the CLI 124
56. Developing application with the Groovy beans DSL 125
57. What to read next 126
VIII. Build tool plugins 127

58. Spring Boot Maven plugin 128
58.1. Including the plugin 128
58.2. Packaging executable jar and war files 129
59. Spring Boot Gradle plugin 130
59.1. Including the plugin 130
59.2. Declaring dependencies without versions 130
Custom version management 131
59.3. Default exclude rules 131
59.4. Packaging executable jar and war files 132
59.5. Running a project in-place 132
59.6. Spring Boot plugin configuration 133
59.7. Repackage configuration 133
59.8. Repackage with custom Gradle configuration 134
Configuration options 134
59.9. Understanding how the Gradle plugin works 135
59.10. Publishing artifacts to a Maven repository using Gradle 135
Configuring Gradle to produce a pom that inherits dependency management 135
Configuring Gradle to produce a pom that imports dependency management 136
60. Supporting other build systems 137
60.1. Repackaging archives 137
60.2. Nested libraries 137
60.3. Finding a main class 137
60.4. Example repackage implementation 137
61. What to read next 138
IX. ‘How-to’ guides 139
62. Spring Boot application 140
62.1. Troubleshoot auto-configuration 140
62.2. Customize the Environment or ApplicationContext before it starts 140
62.3. Build an ApplicationContext hierarchy (adding a parent or root context) 141
62.4. Create a non-web application 141
63. Properties & configuration 142
63.1. Externalize the configuration of SpringApplication 142
63.2. Change the location of external properties of an application 142
63.3. Use ‘short’ command line arguments 142
63.4. Use YAML for external properties 143
63.5. Set the active Spring profiles 143
63.6. Change configuration depending on the environment 144
63.7. Discover built-in options for external properties 144
64. Embedded servlet containers 145
64.1. Add a Servlet, Filter or ServletContextListener to an application 145
64.2. Change the HTTP port 145
64.3. Use a random unassigned HTTP port 145
64.4. Discover the HTTP port at runtime 145
64.5. Configure SSL 146
64.6. Configure Tomcat 146
64.7. Enable Multiple Connectors with Tomcat 146
64.8. Use Tomcat behind a front-end proxy server 147
64.9. Use Jetty instead of Tomcat 147
64.10. Configure Jetty 148
64.11. Use Undertow instead of Tomcat 148

64.12. Configure Undertow 148
64.13. Use Tomcat 7 149
64.14. Use Jetty 8 149
64.15. Create WebSocket endpoints using @ServerEndpoint 150
65. Spring MVC 151
65.1. Write a JSON REST service 151
65.2. Write an XML REST service 151
65.3. Customize the Jackson ObjectMapper 151
65.4. Customize the @ResponseBody rendering 153
65.5. Handling Multipart File Uploads 153
65.6. Switch off the Spring MVC DispatcherServlet 153
65.7. Switch off the Default MVC configuration 153
65.8. Customize ViewResolvers 154
66. Logging 156
66.1. Configure Logback for logging 156
66.2. Configure Log4j for logging 157
67. Data Access 158
67.1. Configure a DataSource 158
67.2. Configure Two DataSources 158
67.3. Use Spring Data repositories 158
67.4. Separate @Entity definitions from Spring configuration 159
67.5. Configure JPA properties 159
67.6. Use a custom EntityManagerFactory 159
67.7. Use Two EntityManagers 159
67.8. Use a traditional persistence.xml 160
67.9. Use Spring Data JPA and Mongo repositories 160
68. Database initialization 161
68.1. Initialize a database using JPA 161
68.2. Initialize a database using Hibernate 161
68.3. Initialize a database using Spring JDBC 161
68.4. Initialize a Spring Batch database 161
68.5. Use a higher level database migration tool 162
Execute Flyway database migrations on startup 162
Execute Liquibase database migrations on startup 162
69. Batch applications 163
69.1. Execute Spring Batch jobs on startup 163
70. Actuator 164
70.1. Change the HTTP port or address of the actuator endpoints 164
70.2. Customize the ‘whitelabel’ error page 164
71. Security 165
71.1. Switch off the Spring Boot security configuration 165
71.2. Change the AuthenticationManager and add user accounts 165
71.3. Enable HTTPS when running behind a proxy server 165
72. Hot swapping 167
72.1. Reload static content 167
72.2. Reload Thymeleaf templates without restarting the container 167
72.3. Reload FreeMarker templates without restarting the container 167
72.4. Reload Groovy templates without restarting the container 167
72.5. Reload Velocity templates without restarting the container 167
72.6. Reload Java classes without restarting the container 167

Configuring Spring Loaded for use with Maven 167
Configuring Spring Loaded for use with Gradle and IntelliJ 168
73. Build 169
73.1. Customize dependency versions with Maven 169
73.2. Create an executable JAR with Maven 169
73.3. Create an additional executable JAR 170
73.4. Extract specific libraries when an executable jar runs 170
73.5. Create a non-executable JAR with exclusions 171
73.6. Remote debug a Spring Boot application started with Maven 172
73.7. Remote debug a Spring Boot application started with Gradle 172
73.8. Build an executable archive with Ant 172
74. Traditional deployment 174
74.1. Create a deployable war file 174
74.2. Create a deployable war file for older servlet containers 174
74.3. Convert an existing application to Spring Boot 174
74.4. Deploying a WAR to Weblogic 175
74.5. Deploying a WAR in an Old (Servlet 2.5) Container 176
X. Appendices 178
A. Common application properties 179
B. Configuration meta-data 187
B.1. Meta-data format 187
Group Attributes 187
Property Attributes 188
Repeated meta-data items 189
B.2. Generating your own meta-data using the annotation processor 189
Nested properties 190
Adding additional meta-data 190
C. Auto-configuration classes 191
C.1. From the “spring-boot-autoconfigure” module 191
C.2. From the “spring-boot-actuator” module 193
D. The executable jar format 194
D.1. Nested JARs 194
The executable jar file structure 194
The executable war file structure 194
D.2. Spring Boot’s “JarFile” class 195
Compatibility with the standard Java “JarFile” 195
D.3. Launching executable jars 195
Launcher manifest 196
Exploded archives 196
D.4. PropertiesLauncher Features 196
D.5. Executable jar restrictions 197
Zip entry compression 197
System ClassLoader 197
D.6. Alternative single jar solutions 197
E. Dependency versions 198

Part I. Spring Boot Documentation

This section provides a brief overview of Spring Boot reference documentation. Think of it as map for the rest of the document. You can read this reference guide in a linear fashion, or you can skip sections if something doesn’t interest you.

The Spring Boot reference guide is available as html, pdf and epub documents. The latest copy is available at docs.spring.io/spring-boot/docs/current/reference.

Having trouble with Spring Boot, We'd like to help!

- Try the How-to's — they provide solutions to the most common questions.
- Learn the Spring basics — Spring Boot is builds on many other Spring projects, check the spring.io web-site for a wealth of reference documentation. If you are just 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 just want to improve them, please get involved.

If you’re just getting started with Spring Boot, or 'Spring' in general, this is the place to start!

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

Ready to actually start using Spring Boot? We've got you covered.

- **Build systems:** Maven | Gradle | Ant | Starter POMs
- **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

Need more details about Spring Boot's core features? This is for you!

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

When you're ready to push your Spring Boot application to production, we've got some tricks that you might like!

- **Management endpoints:** Overview | Customization
- **Connection options:** HTTP | JMX | SSH
- **Monitoring:** Metrics | Auditing | Tracing | Process

Lastly, we have a few topics for the more advanced user.

- **Deploy to the cloud:** Cloud Foundry | Heroku | CloudBees
- **Build tool plugins:** Maven | Gradle
- **Appendix:** Application Properties | Auto-configuration classes | Executable Jars

Part II. Getting started

If you’re just getting started with Spring Boot, or 'Spring' in general, this is the section for you! Here we answer the basic '“what?”', '“how?”' and '“why?”' questions. You’ll find a gentle introduction to Spring Boot along with installation instructions. We’ll then build our first Spring Boot application, discussing some core principles as we go.

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”. We take an opinionated view of the Spring platform and third-party libraries so 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 using `java -jar` or more traditional war deployments. We also provide a command line tool that runs “spring scripts”.

Our primary goals are:

- Provide a radically faster and widely accessible getting started experience for all Spring development.
- Be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).
- Absolutely no code generation and no requirement for XML configuration.

Spring Boot 1.2.0.RELEASE requires Java 6 and Spring Framework 4.1.3 or above. Explicit build support is provided for Maven (3.2+) and Gradle (1.12+).

> **Tip**
>
> Although you can use Spring Boot with Java 6, we generally recommend Java 8 if at all possible.

9.1 Servlet containers

The following embedded servlet containers are supported out of the box:

Name	Servlet Version	Java Version
Tomcat 8	3.1	Java 7+
Tomcat 7	3.0	Java 6+
Jetty 9	3.1	Java 7+
Jetty 8	3.0	Java 6+
Undertow 1.1	3.1	Java 7+

You can also deploy Spring Boot applications to any Servlet 3.0+ compatible container.

Spring Boot can be used with “classic” Java development tools or installed as a command line tool. Regardless, you will need Java SDK v1.6 or higher. You should check your current Java installation before you begin:

```
$ java -version
```

If you are new to Java development, or if you just want to experiment with Spring Boot you might want to try the Spring Boot CLI first, otherwise, read on for “classic” installation instructions.

> **Tip**
>
> Although Spring Boot is compatible with Java 1.6, if possible, you should consider using the latest version of Java.

10.1 Installation instructions for the Java developer

You can use Spring Boot in the same way as any standard Java library. Simply 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; and there is nothing special about a Spring Boot application, so you can run and debug as you would any other Java program.

Although you *could* just 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 3.2 or above. If you don’t already have Maven installed you can follow the instructions at maven.apache.org.

> **Tip**
>
> On many operating systems Maven can be installed via a package manager. If you’re an OSX Homebrew user try `brew install maven`. Ubuntu users can run `sudo apt-get install maven`.

Spring Boot dependencies use the `org.springframework.boot groupId`. Typically your Maven POM file will inherit from the `spring-boot-starter-parent` project and declare dependencies to one or more “Starter POMs”. Spring Boot also provides an optional Maven plugin to create executable jars.

Here is a typical `pom.xml` file:

```
<?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 http://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>

    <!-- Inherit defaults from Spring Boot -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
```

```
        <version>1.2.0.RELEASE</version>
    </parent>

    <!-- Add typical dependencies for a web application -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

    <!-- Package as an executable jar -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

> **Tip**
>
> The `spring-boot-starter-parent` is a great way to use Spring Boot, but it might not be suitable all of the time. Sometimes you may need to inherit from a different parent POM, or you might just not like our default settings. See the section called "Using Spring Boot without the parent POM" for an alternative solution that uses an `import` scope.

Gradle installation

Spring Boot is compatible with Gradle 1.12 or above. If you don't already have Gradle installed you can follow the instructions at www.gradle.org/.

Spring Boot dependencies can be declared using the `org.springframework.boot` `group`. Typically your project will declare dependencies to one or more "Starter POMs". 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's a small script and library that you commit alongside your code to bootstrap the build process. See www.gradle.org/docs/current/userguide/gradle_wrapper.html for details.

Here is a typical `build.gradle` file:

```
buildscript {
    repositories {
        jcenter()
        maven { url "http://repo.spring.io/snapshot" }
        maven { url "http://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.2.0.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

jar {
```

```
    baseName = 'myproject'
    version =  '0.0.1-SNAPSHOT'
}

repositories {
    jcenter()
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-test")
}
```

10.2 Installing the Spring Boot CLI

The Spring Boot CLI is a command line tool that can be used if you want to quickly prototype with Spring. It allows you to run Groovy scripts, which means that you have a familiar Java-like syntax, without so much boilerplate code.

You don't need to use the CLI to work with Spring Boot but it's definitely the quickest way to get a Spring application off the ground.

Manual installation

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

- spring-boot-cli-1.2.0.RELEASE-bin.zip
- spring-boot-cli-1.2.0.RELEASE-bin.tar.gz

Cutting edge snapshot distributions are also available.

Once downloaded, follow the INSTALL.txt instructions from the unpacked archive. In summary: there is a `spring` script (`spring.bat` for Windows) in a `bin/` directory in the `.zip` file, or 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 GVM

GVM (the Groovy Environment Manager) can be used for managing multiple versions of various Groovy and Java binary packages, including Groovy itself and the Spring Boot CLI. Get `gvm` from gvmtool.net and install Spring Boot with

```
$ gvm install springboot
$ spring --version
Spring Boot v1.2.0.RELEASE
```

If you are developing features for the CLI and want easy access to the version you just built, follow these extra instructions.

```
$ gvm install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-cli-1.2.0.RELEASE-
bin/spring-1.2.0.RELEASE/
$ gvm use springboot dev
$ spring --version
Spring CLI v1.2.0.RELEASE
```

This will install a local instance of `spring` called the `dev` instance inside your gvm repository. It points at your target build location, so every time you rebuild Spring Boot, `spring` will be up-to-date.

You can see it by doing this:

```
$ gvm ls springboot

================================================================================
Available Springboot Versions
================================================================================
> + dev
* 1.2.0.RELEASE

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

OSX Homebrew installation

If you are on a Mac and using Homebrew, all you need to do to install the Spring Boot CLI is:

```
$ brew tap pivotal/tap
$ brew install springboot
```

Homebrew will install `spring` to `/usr/local/bin`.

> **Note**
>
> If you don't see the formula, your installation of brew might be out-of-date. Just execute `brew update` and try again.

Command-line completion

Spring Boot CLI ships with scripts that provide command completion for BASH and zsh shells. You can `source` the script (also named `spring`) in any shell, or put it in your personal or system-wide bash completion initialization. On a Debian system the system-wide scripts are in `/shell-completion/bash` and all scripts in that directory are executed when a new shell starts. To run the script manually, e.g. if you have installed using `GVM`

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

> **Note**
>
> If you install Spring Boot CLI using Homebrew, the command-line completion scripts are automatically registered with your shell.

Quick start Spring CLI example

Here's a really simple web application that you can use to test your installation. Create a file called `app.groovy`:

```
@RestController
class ThisWillActuallyRun {

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

}
```

Then simply run it from a shell:

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

> **Note**
>
> It will take some time when you first run the application as dependencies are downloaded. Subsequent runs will be much quicker.

Open localhost:8080 in your favorite web browser and you should see the following output:

```
Hello World!
```

10.3 Upgrading from an earlier version of Spring Boot

If you are upgrading from an earlier release of Spring Boot check the “release notes” hosted on the project wiki. You’ll find upgrade instructions along with a list of “new and noteworthy” features for each release.

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

Let's develop a simple "Hello World!" web application in Java that highlights some of Spring Boot's key features. We'll use Maven to build this project since most IDEs support it.

> **Tip**
>
> The spring.io web site contains many "Getting Started" guides that use Spring Boot. If you're looking to solve a specific problem; check there first.

Before we begin, open a terminal to check that you have valid versions of Java and Maven installed.

```
$ java -version
java version "1.7.0_51"
Java(TM) SE Runtime Environment (build 1.7.0_51-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)
```

```
$ mvn -v
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-11T13:58:10-07:00)
Maven home: /Users/user/tools/apache-maven-3.1.1
Java version: 1.7.0_51, vendor: Oracle Corporation
```

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

11.1 Creating the POM

We need to start by creating a Maven `pom.xml` file. The `pom.xml` is the recipe that will be 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 http://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>1.2.0.RELEASE</version>
    </parent>

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

</project>
```

This should give you a working build, you can test it out by running `mvn package` (you can ignore the "jar will be empty - no content was marked for inclusion!" warning for now).

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

11.2 Adding classpath dependencies

Spring Boot provides a number of “Starter POMs” that make easy to add jars to your classpath. Our sample application has already used `spring-boot-starter-parent` in the `parent` section of the POM. The `spring-boot-starter-parent` is a special starter that provides useful Maven defaults. It also provides a `dependency-management` section so that you can omit `version` tags for “blessed” dependencies.

Other “Starter POMs” simply provide dependencies that you are likely to need when developing a specific type of application. Since we are developing a web application, we will add a `spring-boot-starter-web` dependency — but before that, let’s look at what we currently have.

```
$ 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. Let’s edit our `pom.xml` and add the `spring-boot-starter-web` dependency just 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 will see that there are now a number of additional dependencies, including the Tomcat web server and Spring Boot itself.

11.3 Writing the code

To finish our application we need to create a single Java file. Maven will compile sources from `src/main/java` by default so you need to create that folder structure, then add a file named `src/main/java/Example.java`:

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

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

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

}
```

Although there isn’t much code here, quite a lot is going on. Let’s step through the important parts.

The @RestController and @RequestMapping annotations

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

The `@RequestMapping` annotation provides "routing" information. It is telling 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 @EnableAutoConfiguration annotation

The second class-level annotation is `@EnableAutoConfiguration`. This annotation tells Spring Boot to "guess" how you will 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 will assume that you are developing a web application and setup Spring accordingly.

> **Starter POMs and Auto-Configuration**
>
> Auto-configuration is designed to work well with "Starter POMs", but the two concepts are not directly tied. You are free to pick-and-choose jar dependencies outside of the starter POMs and Spring Boot will still do its best to auto-configure your application.

The "main" method

The final part of our application is the `main` method. This is just 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` will bootstrap our application, starting Spring which will in turn start the auto-configured Tomcat web server. We need to pass `Example.class` as an argument to the `run` method to tell `SpringApplication` which is the primary Spring component. The `args` array is also passed through to expose any command-line arguments.

11.4 Running the example

At this point our application should work. Since we have used the `spring-boot-starter-parent` POM we have a useful `run` goal that we can use to start the application. Type `mvn spring-boot:run` from the root project directory to start the application:

```
$ mvn spring-boot:run

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
```

```
 :: Spring Boot ::  (v1.2.0.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.514)
```

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

```
Hello World!
```

To gracefully exit the application hit `ctrl-c`.

11.5 Creating an executable jar

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

Executable jars and Java

Java does not provide any standard way to load nested jar files (i.e. 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 "shaded" jars. A shaded jar simply packages all classes, from all jars, into a single "uber jar". The problem with shaded jars is that it becomes hard to see which libraries you are actually using in your application. It can also be problematic if the the same filename is used (but with different content) in multiple jars.

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

To create an executable jar we need to add the `spring-boot-maven-plugin` to our `pom.xml`. 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 are not using the parent POM you will need to declare this configuration yourself. See the plugin documentation for details.

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

```
$ 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:1.2.0.RELEASE: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 10 Mb in size. If you want to peek inside, you can use `jar tvf`:

```
$ 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:

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

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.2.0.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.536 seconds (JVM running for 2.864)
```

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

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

Otherwise, the next logical step is to read *Part III, "Using Spring Boot"*. If you're really impatient, you could also jump ahead and read about *Spring Boot features*.

Part III. Using Spring Boot

This section goes into more detail about how you should use Spring Boot. It covers topics such as build systems, auto-configuration and run/deployment options. 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, will make your development process just a little easier.

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

It is strongly recommended that you choose a build system that supports *dependency management*, and one 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 will not be particularly well supported.

13.1 Maven

Maven users can inherit from the `spring-boot-starter-parent` project to obtain sensible defaults. The parent project provides the following features:

- Java 1.6 as the default compiler level.
- UTF-8 source encoding.
- A Dependency Management section, allowing you to omit `<version>` tags for common dependencies, inherited from the `spring-boot-dependencies` POM.
- Sensible resource filtering.
- Sensible plugin configuration (exec plugin, surefire, Git commit ID, shade).
- Sensible resource filtering for `application.properties` and `application.yml`

On the last point: since the default config files files accept Spring style placeholders (`${…}`) the Maven filtering is changed to use `@..@` placeholders (you can override that with a Maven property `resource.delimiter`).

Inheriting the starter parent

To configure your project to inherit from the `spring-boot-starter-parent` simply set the `parent`:

```
<!-- Inherit defaults from Spring Boot -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.0.RELEASE</version>
</parent>
```

> **Note**
>
> You should only need to specify the Spring Boot version number on this dependency. If you import additional starters, you can safely omit the version number.

Using Spring Boot without the parent POM

Not everyone likes inheriting from the `spring-boot-starter-parent` POM. You may have your own corporate standard parent that you need to use, or you may just prefer to explicitly declare all your Maven configuration.

If you don’t want to use the `spring-boot-starter-parent`, you can still keep the benefit of the dependency management (but not the plugin management) by using a `scope=import` dependency:

```
<dependencyManagement>
     <dependencies>
        <dependency>
            <!-- Import dependency management from Spring Boot -->
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
```

```
            <version>1.2.0.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Changing the Java version

The `spring-boot-starter-parent` chooses fairly conservative Java compatibility. If you want to follow our recommendation and use a later Java version you can add a `java.version` property:

```
<properties>
    <java.version>1.8</java.version>
</properties>
```

Using the Spring Boot Maven plugin

Spring Boot includes a Maven plugin that can package the project as an executable jar. Add the plugin to your `<plugins>` section if you want to use it:

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

> **Note**
>
> If you use the Spring Boot starter parent pom, you only need to add the plugin, there is no need for to configure it unless you want to change the settings defined in the parent.

13.2 Gradle

Gradle users can directly import “starter POMs” in their `dependencies` section. Unlike Maven, there is no “super parent” to import to share some configuration.

```
apply plugin: 'java'

repositories { jcenter() }
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.2.0.RELEASE")
}
```

The `spring-boot-gradle-plugin` is also available and provides tasks to create executable jars and run projects from source. It also adds a `ResolutionStrategy` that enables you to omit the version number for “blessed” dependencies:

```
buildscript {
    repositories { jcenter() }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.2.0.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

repositories { jcenter() }
```

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-test")
}
```

13.3 Ant

It is possible to build a Spring Boot project using Apache Ant, however, no special support or plugins are provided. Ant scripts can use the Ivy dependency system to import starter POMs.

See the *Section 73.8, “Build an executable archive with Ant”* “How-to” for more complete instructions.

13.4 Starter POMs

Starter POMs 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 technology 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, just include the `spring-boot-starter-data-jpa` dependency in your project, and you are good to go.

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

What’s in a name

All 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 allow you to search dependencies by name. For example, with the appropriate Eclipse or STS plugin installed, you can simply hit `ctrl-space` in the POM editor and type "spring-boot-starter" for a complete list.

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

Table 13.1. Spring Boot application starters

Name	Description
`spring-boot-starter`	The core Spring Boot starter, including auto-configuration support, logging and YAML.
`spring-boot-starter-actuator`	Production ready features to help you monitor and manage your application.
`spring-boot-starter-amqp`	Support for the “Advanced Message Queuing Protocol” via `spring-rabbit`.
`spring-boot-starter-aop`	Support for aspect-oriented programming including `spring-aop` and AspectJ.
`spring-boot-starter-batch`	Support for “Spring Batch” including HSQLDB database.
`spring-boot-starter-cloud-connectors`	Support for “Spring Cloud Connectors” which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku.

Name	Description
`spring-boot-starter-data-elasticsearch`	Support for the Elasticsearch search and analytics engine including `spring-data-elasticsearch`.
`spring-boot-starter-data-gemfire`	Support for the GemFire distributed data store including `spring-data-gemfire`.
`spring-boot-starter-data-jpa`	Support for the “Java Persistence API” including `spring-data-jpa`, `spring-orm` and Hibernate.
`spring-boot-starter-data-mongodb`	Support for the MongoDB NoSQL Database, including `spring-data-mongodb`.
`spring-boot-starter-data-rest`	Support for exposing Spring Data repositories over REST via `spring-data-rest-webmvc`.
`spring-boot-starter-data-solr`	Support for the Apache Solr search platform, including `spring-data-solr`.
`spring-boot-starter-freemarker`	Support for the FreeMarker templating engine
`spring-boot-starter-groovy-templates`	Support for the Groovy templating engine
`spring-boot-starter-hornetq`	Support for “Java Message Service API” via HornetQ.
`spring-boot-starter-integration`	Support for common `spring-integration` modules.
`spring-boot-starter-jdbc`	Support for JDBC databases.
`spring-boot-starter-jersey`	Support for the Jersey RESTful Web Services framework.
`spring-boot-starter-jta-atomikos`	Support for JTA distributed transactions via Atomikos.
`spring-boot-starter-jta-bitronix`	Support for JTA distributed transactions via Bitronix.
`spring-boot-starter-mail`	Support for `javax.mail`.
`spring-boot-starter-mobile`	Support for `spring-mobile`
`spring-boot-starter-redis`	Support for the REDIS key-value data store, including `spring-redis`.
`spring-boot-starter-security`	Support for `spring-security`.
`spring-boot-starter-social-facebook`	Support for `spring-social-facebook`.
`spring-boot-starter-social-linkedin`	Support for `spring-social-linkedin`.
`spring-boot-starter-social-twitter`	Support for `spring-social-twitter`.

Name	Description
`spring-boot-starter-test`	Support for common test dependencies, including JUnit, Hamcrest and Mockito along with the `spring-test` module.
`spring-boot-starter-thymeleaf`	Support for the Thymeleaf templating engine, including integration with Spring.
`spring-boot-starter-velocity`	Support for the Velocity templating engine
`spring-boot-starter-web`	Support for full-stack web development, including Tomcat and `spring-webmvc`.
`spring-boot-starter-websocket`	Support for WebSocket development.
`spring-boot-starter-ws`	Support for Spring Web Services

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

Table 13.2. Spring Boot production ready starters

Name	Description
`spring-boot-starter-actuator`	Adds production ready features such as metrics and monitoring.
`spring-boot-starter-remote-shell`	Adds remote `ssh` shell support.

Finally, Spring Boot includes some starters that can be used if you want to exclude or swap specific technical facets.

Table 13.3. Spring Boot technical starters

Name	Description
`spring-boot-starter-jetty`	Imports the Jetty HTTP engine (to be used as an alternative to Tomcat)
`spring-boot-starter-log4j`	Support the Log4J logging framework
`spring-boot-starter-logging`	Import Spring Boot's default logging framework (Logback).
`spring-boot-starter-tomcat`	Import Spring Boot's default HTTP engine (Tomcat).
`spring-boot-starter-undertow`	Imports the Undertow HTTP engine (to be used as an alternative to Tomcat)

> **Tip**
>
> For a list of additional community contributed starter POMs, see the README file in the `spring-boot-starters` module on GitHub.

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

14.1 Using the “default” package

When a class doesn't 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 `@ComponentScan`, `@EntityScan` or `@SpringBootApplication` annotations, since every class from every jar, will be read.

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

14.2 Locating the main application class

We generally recommend that you locate your main application class in a root package above other classes. The `@EnableAutoConfiguration` 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 `@EnableAutoConfiguration` annotated class will be used to search for `@Entity` items.

Using a root package also allows the `@ComponentScan` annotation to be used without needing to specify a `basePackage` attribute. You can also use the `@SpringBootApplication` annotation if your main class is in the root package.

Here is a typical layout:

```
com
 +- example
     +- myproject
         +- Application.java
         |
         +- domain
         |   +- Customer.java
         |   +- CustomerRepository.java
         |
         +- service
         |   +- CustomerService.java
         |
         +- web
             +- CustomerController.java
```

The `Application.java` file would declare the `main` method, along with the basic `@Configuration`.

```
package com.example.myproject;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {
```

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

}
```

Spring Boot favors Java-based configuration. Although it is possible to call `SpringApplication.run()` with an XML source, we generally recommend that your primary source is a `@Configuration` class. Usually the class that defines the `main` method is also a good candidate as the primary `@Configuration`.

> **Tip**
>
> Many Spring configuration examples have been published on the Internet that use XML configuration. Always try to use the equivalent Java-base configuration if possible. Searching for `enable*` annotations can be a good starting point.

15.1 Importing additional configuration classes

You don't need to 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 pickup all Spring components, including `@Configuration` classes.

15.2 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 additional `@ImportResource` annotation to load XML configuration files.

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 we will auto-configure 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 `@EnableAutoConfiguration` annotation. We generally recommend that you add it to your primary `@Configuration` class.

16.1 Gradually replacing auto-configuration

Auto-configuration is noninvasive, 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 will back away.

If you need to find out what auto-configuration is currently being applied, and why, starting your application with the `--debug` switch. This will log an auto-configuration report to the console.

16.2 Disabling specific auto-configuration

If you find that specific auto-configure classes are being applied that you don't want, you can use the exclude attribute of `@EnableAutoConfiguration` to disable them.

```
import org.springframework.boot.autoconfigure.*;
import org.springframework.boot.autoconfigure.jdbc.*;
import org.springframework.context.annotation.*;

@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. For simplicity, we often find that using `@ComponentScan` to find your beans, in combination with `@Autowired` constructor injection works well.

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

Here is an example `@Service` Bean that uses constructor injection to obtain a required `RiskAssessor` bean.

```
package com.example.service;

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

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...

}
```

> **Tip**
>
> Notice how using constructor injection allows the `riskAssessor` field to be marked as `final`, indicating that it cannot be subsequently changed.

Many Spring Boot developers always have their main class annotated with `@Configuration`, `@EnableAutoConfiguration` and `@ComponentScan`. Since these annotations are so frequently used together (especially if you follow the best practices above), Spring Boot provides a convenient `@SpringBootApplication` alternative.

The `@SpringBootApplication` annotation is equivalent to using `@Configuration`, `@EnableAutoConfiguration` and `@ComponentScan` with their default attributes:

```
package com.example.myproject;

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

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
public class Application {

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

}
```

One of the biggest advantages of packaging your application as jar and using an embedded HTTP server is that you can run your application as you would any other. Debugging Spring Boot applications is also easy; you don't 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 you should refer to your server and IDE documentation.

19.1 Running from an IDE

You can run a Spring Boot application from your IDE as a simple Java application, however, first you will need to import your project. Import steps will 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 can't directly import your project into your IDE, you may be able to generate IDE metadata 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 will see a "Port already in use" error. STS users can use the `Relaunch` button rather than `Run` to ensure that any existing instance is closed.

19.2 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`. For example:

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

It is also possible to run a packaged application with remote debugging support enabled. This allows you to attach a debugger to your packaged application:

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

19.3 Using the Maven plugin

The Spring Boot Maven plugin includes a `run` goal which can be used to quickly compile and run your application. Applications run in an exploded form, and you can edit resources for instant "hot" reload.

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

You might also want to use the useful operating system environment variable:

```
$ export MAVEN_OPTS=-Xmx1024m -XX:MaxPermSize=128M -Djava.security.egd=file:/dev/./urandom
```

(The "egd" setting is to speed up Tomcat startup by giving it a faster source of entropy for session keys.)

19.4 Using the Gradle plugin

The Spring Boot Gradle plugin also includes a `run` goal which can be used to run your application in an exploded form. The `bootRun` task is added whenever you import the `spring-boot-plugin`

```
$ gradle bootRun
```

You might also want to use this useful operating system environment variable:

```
$ export JAVA_OPTS=-Xmx1024m -XX:MaxPermSize=128M -Djava.security.egd=file:/dev/./urandom
```

19.5 Hot swapping

Since Spring Boot applications are just 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 the Spring Loaded project, or JRebel can be used.

See the Hot swapping “How-to” section for details.

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

For additional “production ready” features, such as health, auditing and metric REST or JMX end-points; consider adding `spring-boot-actuator`. See *Part V, “Spring Boot Actuator: Production-ready features”* for details.

You should now have good understanding of how you can use Spring Boot along with 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.

Part IV. Spring Boot features

This section dives into the details of Spring Boot. Here you can learn about the key features that you will want to use and customize. If you haven't already, you might want to read the *Part II, "Getting started"* and *Part III, "Using Spring Boot"* sections so that you have a good grounding of the basics.

The `SpringApplication` class provides a convenient way to bootstrap a Spring application that will be started from a `main()` method. In many situations you can just delegate to the static `SpringApplication.run` method:

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

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

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v1.2.0.RELEASE

2013-07-31 00:08:16.117  INFO 56603 --- [           main] o.s.b.s.app.SampleApplication            :
 Starting SampleApplication v0.1.0 on mycomputer with PID 56603 (/apps/myapp.jar started by pwebb)
2013-07-31 00:08:16.166  INFO 56603 --- [           main] ationConfigEmbeddedWebApplicationContext :
 Refreshing
 org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@6e5a8246:
 startup date [Wed Jul 31 00:08:16 PDT 2013]; root of context hierarchy
2014-03-04 13:09:54.912  INFO 41370 --- [           main] .t.TomcatEmbeddedServletContainerFactory :
 Server initialized with port: 8080
2014-03-04 13:09:56.501  INFO 41370 --- [           main] o.s.b.s.app.SampleApplication            :
 Started SampleApplication in 2.992 seconds (JVM running for 3.658)
```

By default `INFO` logging messages will be shown, including some relevant startup details such as the user that launched the application.

22.1 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 `banner.location` to the location of such a file. If the file has an unusual encoding you can set `banner.encoding` (default is `UTF-8`).

You can use the following variables inside your `banner.txt` file:

Table 22.1. Banner variables

Variable	Description
`${application.version}`	The version number of your application as declared in `MANIFEST.MF`. For example `1.0`.
`${application.formatted-version}`	The version number of your application as declared in `MANIFEST.MF` formatted for display (surrounded with brackets and prefixed with `v`). For example `(v1.0)`.
`${spring-boot.version}`	The Spring Boot version that you are using. For example `1.2.0.RELEASE`.
`${spring-boot.formatted-version}`	The Spring Boot version that you are using formatted for display (surrounded with brackets and prefixed with `v`). For example `(v1.2.0.RELEASE)`.

> **Tip**
>
> The `SpringBootApplication.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.

22.2 Customizing SpringApplication

If the `SpringApplication` defaults aren't to your taste you can instead create a local instance and customize it. For example, to turn off the banner you would write:

```
public static void main(String[] args) {
    SpringApplication app = new SpringApplication(MySpringConfiguration.class);
    app.setShowBanner(false);
    app.run(args);
}
```

> **Note**
>
> The constructor arguments passed to `SpringApplication` are configuration sources for spring beans. In most cases these will be references to `@Configuration` classes, but they could also be references to XML configuration or to packages that should be scanned.

It is also possible to configure the `SpringApplication` using an `application.properties` file. See *Chapter 23, Externalized Configuration* for details.

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

22.3 Fluent builder API

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

The `SpringApplicationBuilder` allows you to chain together multiple method calls, and includes `parent` and `child` methods that allow you to create a hierarchy.

For example:

```
new SpringApplicationBuilder()
    .showBanner(false)
    .sources(Parent.class)
    .child(Application.class)
    .run(args);
```

> **Note**
>
> There are some restrictions when creating an `ApplicationContext` hierarchy, e.g. Web components **must** be contained within the child context, and the same `Environment` will be used for both parent and child contexts. See the `SpringApplicationBuilder` javadoc for full details.

22.4 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.

You can register event listeners in a number of ways, the most common being `SpringApplication.addListeners(…)` method.

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

1. An `ApplicationStartedEvent` is sent at the start of a run, but before any processing except 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 `ApplicationPreparedEvent` is sent just before the refresh is started, but after bean definitions have been loaded.
4. An `ApplicationFailedEvent` is sent if there is an exception on startup.

> **Tip**
>
> You often won't need to use application events, but it can be handy to know that they exist. Internally, Spring Boot uses events to handle a variety of tasks.

22.5 Web environment

A `SpringApplication` will attempt to create the right type of `ApplicationContext` on your behalf. By default, an `AnnotationConfigApplicationContext` or `AnnotationConfigEmbeddedWebApplicationContext` will be used, depending on whether you are developing a web application or not.

The algorithm used to determine a 'web environment' is fairly simplistic (based on the presence of a few classes). You can use `setWebEnvironment(boolean webEnvironment)` if you need to override the default.

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

> **Tip**
>
> It is often desirable to call `setWebEnvironment(false)` when using `SpringApplication` within a JUnit test.

22.6 Using the CommandLineRunner

If you want access to the raw command line arguments, or you need to run some specific code once the `SpringApplication` has started you can implement the `CommandLineRunner` interface. The `run(String… args)` method will be called on all Spring beans implementing this interface.

```
import org.springframework.boot.*
import org.springframework.stereotype.*

@Component
public class MyBean implements CommandLineRunner {

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

}
```

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

22.7 Application exit

Each `SpringApplication` will register a shutdown hook with the JVM to ensure that the `ApplicationContext` is closed 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 the application ends.

Spring Boot allows you to externalize your configuration so you can work with the same application code in different environments. You can use properties files, YAML files, environment variables and command-line arguments to externalize configuration. Property values can be injected directly into your beans using the `@Value` annotation, accessed via Spring's `Environment` abstraction or bound to structured objects.

Spring Boot uses a very particular `PropertySource` order that is designed to allow sensible overriding of values, properties are considered in the the following order:

1. Command line arguments.
2. JNDI attributes from `java:comp/env`.
3. Java System properties (`System.getProperties()`).
4. OS environment variables.
5. A `RandomValuePropertySource` that only has properties in `random.*`.
6. Application properties outside of your packaged jar (`application.properties` including YAML and profile variants).
7. Application properties packaged inside your jar (`application.properties` including YAML and profile variants).
8. `@PropertySource` annotations on your `@Configuration` classes.
9. Default properties (specified using `SpringApplication.setDefaultProperties`).

To provide a concrete example, suppose you develop a `@Component` that uses a `name` property:

```
import org.springframework.stereotype.*
import org.springframework.beans.factory.annotation.*

@Component
public class MyBean {

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

    // ...

}
```

You can bundle an `application.properties` inside your jar that provides a sensible default `name`. When running in production, an `application.properties` can be provided outside of your jar that overrides `name`; and for one-off testing, you can launch with a specific command line switch (e.g. `java -jar app.jar --name="Spring"`).

The `RandomValuePropertySource` is useful for injecting random values (e.g. into secrets or test cases). It can produce integers, longs or strings, e.g.

```
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.number.less.than.ten=${random.int(10)}
my.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 (exclusive).

23.1 Accessing command line properties

By default `SpringApplication` will convert any command line option arguments (starting with '--', e.g. `--server.port=9000`) to a `property` and add it to the Spring `Environment`. As mentioned above, command line properties always take precedence over other property sources.

If you don't want command line properties to be added to the `Environment` you can disable them using `SpringApplication.setAddCommandLineProperties(false)`.

23.2 Application property files

`SpringApplication` will load properties from `application.properties` files in the following locations and add them to the Spring `Environment`:

1. A `/config` subdir of the current directory.
2. The current directory
3. A classpath `/config` package
4. The classpath root

The list is ordered by precedence (locations higher in the list override lower items).

> **Note**
>
> You can also use YAML ('.yml') files as an alternative to '.properties'.

If you don't like `application.properties` as the configuration file name you can switch to another by specifying a `spring.config.name` environment property. You can also refer to an explicit location using the `spring.config.location` environment property (comma-separated list of directory locations, or file paths).

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

or

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

If `spring.config.location` contains directories (as opposed to files) they should end in `/` (and will be appended with the names generated from `spring.config.name` before being loaded). The default search path `classpath:,classpath:/config,file:,file:config/` is always used, irrespective of the value of `spring.config.location`. In that way you can set up default values for your application in `application.properties` (or whatever other basename you choose with `spring.config.name`) and override it at runtime with a different file, keeping the defaults.

> **Note**
>
> If you use environment variables rather than system properties, most operating systems disallow period-separated key names, but you can use underscores instead (e.g. `SPRING_CONFIG_NAME` instead of `spring.config.name`).

> **Note**
>
> If you are running in a container 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.

23.3 Profile specific properties

In addition to `application.properties` files, profile specific properties can also be defined using the naming convention `application-{profile}.properties`.

Profile specific properties are loaded from the same locations as standard `application.properties`, with profile specific files overriding the default ones.

23.4 Placeholders in properties

The values in `application.properties` are filtered through the existing `Environment` when they are used so you can refer back to previously defined values (e.g. from System properties).

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

> **Tip**
>
> You can also use this technique to create 'short' variants of existing Spring Boot properties. See the *Section 63.3, "Use 'short' command line arguments"* how-to for details.

23.5 Using YAML instead of Properties

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

> **Note**
>
> If you use 'starter POMs' SnakeYAML will be automatically provided via `spring-boot-starter`.

Loading YAML

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

For example, the following YAML document:

```
environments:
    dev:
        url: http://dev.bar.com
        name: Developer Setup
    prod:
        url: http://foo.bar.com
        name: My Cool App
```

Would be transformed into these properties:

```
environments.dev.url=http://dev.bar.com
environments.dev.name=Developer Setup
environments.prod.url=http://foo.bar.com
environments.prod.name=My Cool App
```

YAML lists are represented as property keys with `[index]` dereferencers, for example this YAML:

```
my:
   servers:
       - dev.bar.com
       - foo.bar.com
```

Would be transformed into these properties:

```
my.servers[0]=dev.bar.com
my.servers[1]=foo.bar.com
```

To bind to properties like that using the Spring `DataBinder` utilities (which is what `@ConfigurationProperties` does) you need to have a property in the target bean of type `java.util.List` (or `Set`) and you either need to provide a setter, or initialize it with a mutable value, e.g. this will bind to the properties above

```
@ConfigurationProperties(prefix="my")
public class Config {

    private List<String> servers = new ArrayList<String>();

    public List<String> getServers() {
        return this.servers;
    }
}
```

Exposing YAML as properties in the Spring Environment

The `YamlPropertySourceLoader` class can be used to expose YAML as a `PropertySource` in the Spring `Environment`. This allows you to use the familiar `@Value` annotation with placeholders syntax to access YAML properties.

Multi-profile YAML documents

You can specify multiple profile-specific YAML documents in a single file by using a `spring.profiles` key to indicate when the document applies. For example:

```
server:
    address: 192.168.1.100
---
spring:
    profiles: development
server:
    address: 127.0.0.1
---
spring:
    profiles: production
```

```
server:
    address: 192.168.1.120
```

In the example above, the `server.address` property will be `127.0.0.1` if the `development` profile is active. If the `development` and `production` profiles are **not** enabled, then the value for the property will be `192.168.1.100`

YAML shortcomings

YAML files can't be loaded via the `@PropertySource` annotation. So in the case that you need to load values that way, you need to use a properties file.

23.6 Typesafe 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 allows strongly typed beans to govern and validate the configuration of your application. For example:

```
@Component
@ConfigurationProperties(prefix="connection")
public class ConnectionSettings {

    private String username;

    private InetAddress remoteAddress;

    // ... getters and setters

}
```

When the `@EnableConfigurationProperties` annotation is applied to your `@Configuration`, any beans annotated with `@ConfigurationProperties` will be automatically configured from the `Environment` properties. This style of configuration works particularly well with the `SpringApplication` external YAML configuration:

```
# application.yml

connection:
    username: admin
    remoteAddress: 192.168.1.1

# additional configuration as required
```

To work with `@ConfigurationProperties` beans you can just inject them in the same way as any other bean.

```
@Service
public class MyService {

    @Autowired
    private ConnectionSettings connection;

     //...

    @PostConstruct
    public void openConnection() {
        Server server = new Server();
        this.connection.configure(server);
    }

}
```

It is also possible to shortcut the registration of `@ConfigurationProperties` bean definitions by simply listing the properties classes directly in the `@EnableConfigurationProperties` annotation:

```
@Configuration
@EnableConfigurationProperties(ConnectionSettings.class)
public class MyConfiguration {
}
```

> **Tip**
>
> Using `@ConfigurationProperties` also allows you to generate meta-data files that can be used by IDEs. See the Appendix B, *Configuration meta-data* appendix for details.

Relaxed binding

Spring Boot uses some relaxed rules for binding `Environment` properties to `@ConfigurationProperties` beans, so there doesn't need to be an exact match between the `Environment` property name and the bean property name. Common examples where this is useful include underscore separated (e.g. `context_path` binds to `contextPath`), and capitalized (e.g. `PORT` binds to `port`) environment properties.

Spring will attempt 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 bean id `conversionService`) or custom property editors (via a `CustomEditorConfigurer` bean).

@ConfigurationProperties Validation

Spring Boot will attempt to validate external configuration, by default using JSR-303 (if it is on the classpath). You can simply add JSR-303 `javax.validation` constraint annotations to your `@ConfigurationProperties` class:

```
@Component
@ConfigurationProperties(prefix="connection")
public class ConnectionSettings {

    @NotNull
    private InetAddress remoteAddress;

    // ... getters and setters

}
```

You can also add a custom Spring `Validator` by creating a bean definition called `configurationPropertiesValidator`.

> **Tip**
>
> The `spring-boot-actuator` module includes an endpoint that exposes all `@ConfigurationProperties` beans. Simply point your web browser to `/configprops` or use the equivalent JMX endpoint. See the *Production ready features*. section for details.

Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments. Any `@Component` or `@Configuration` can be marked with `@Profile` to limit when it is loaded:

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```

In the normal Spring way, you can use a `spring.profiles.active` `Environment` property to specify which profiles are active. You can specify the property in any of the usual ways, for example you could include it in your `application.properties`:

```
spring.profiles.active=dev,hsqldb
```

or specify on the command line using the switch `--spring.profiles.active=dev,hsqldb`.

24.1 Adding active profiles

The `spring.profiles.active` property follows the same ordering rules as other properties, the highest `PropertySource` will win. This means that you can specify active profiles in `application.properties` then **replace** them using the command line switch.

Sometimes it is useful to have profile specific properties that **add** to the active profiles rather than replace them. The `spring.profiles.include` property can be used to unconditionally add active profiles. The `SpringApplication` entry point also has a Java API for setting additional profiles (i.e. on top of those activated by the `spring.profiles.active` property): see the `setAdditionalProfiles()` method.

For example, when an application with following properties is run using the switch `--spring.profiles.active=prod` the `proddb` and `prodmq` profiles will also be activated:

```
---
my.property: fromyamlfile
---
spring.profiles: prod
spring.profiles.include: proddb,prodmq
```

> **Note**
>
> Remember that the `spring.profiles` property can be defined in a YAML document to determine when this particular document is included in the configuration. See Section 63.6, “Change configuration depending on the environment” for more details.

24.2 Programmatically setting profiles

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

24.3 Profile specific configuration files

Profile specific variants of both `application.properties` (or `application.yml`) and files referenced via `@ConfigurationProperties` are considered as files are loaded. See *Section 23.3, "Profile specific properties"* for details.

Spring Boot uses Commons Logging for all internal logging, but leaves the underlying log implementation open. Default configurations are provided for Java Util Logging, Log4J, 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 'Starter POMs', Logback will be used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J or SLF4J will all work correctly.

> **Tip**
>
> There are a lot of logging frameworks available for Java. Don't worry if the above list seems confusing. Generally you won't need to change your logging dependencies and the Spring Boot defaults will work just fine.

25.1 Log format

The default log output from Spring Boot looks like this:

```
2014-03-05 10:57:51.112  INFO 45469 --- [           main] org.apache.catalina.core.StandardEngine  :
 Starting Servlet Engine: Apache Tomcat/7.0.52
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       :
 Initializing Spring embedded WebApplicationContext
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.s.web.context.ContextLoader            :
 Root WebApplicationContext: initialization completed in 1358 ms
2014-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean        :
 Mapping servlet: 'dispatcherServlet' to [/]
2014-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  :
 Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
```

The following items are output:

- Date and Time — Millisecond precision and easily sortable.
- Log Level — `ERROR`, `WARN`, `INFO`, `DEBUG` or `TRACE`.
- Process ID.
- A `---` separator to distinguish the start of actual log messages.
- Logger name — This is usually the source class name (often abbreviated).
- The log message.

25.2 Console output

The default log configuration will echo messages to the console as they are written. By default `ERROR`, `WARN` and `INFO` level messages are logged. To also log `DEBUG` level messages to the console you can start your application with a `--debug` flag.

```
$ java -jar myapp.jar --debug
```

If your terminal supports ANSI, color output will be used to aid readability. You can set `spring.output.ansi.enabled` to a supported value to override the auto detection.

25.3 File output

By default, Spring Boot will only log to the console and will not write log files. If you want to write log files in addition to the console output you need to set the `logging.file` and/or `logging.path` properties (for example in your `application.properties`). Log files will rotate when they reach 10 Mb.

As with console output, `ERROR`, `WARN` and `INFO` level messages are logged by default.

The following table shows how the `logging.*` properties can be used together:

Table 25.1. Logging properties

logging.path	logging.file	Example	Description
(none)	Exact location	`./my.log`	Writes to the specified file.
(none)	Simple name	`my.log`	Writes the given file in the `temp` folder.
Specific folder	Simple name	`/logs` & `my.log`	Writes the given file in the specified folder.
Specific folder	*(none)*	`/logs`	Writes the `spring.log` in the specified folder.

25.4 Log Levels

All the supported logging systems can have the logger levels set in the Spring `Environment` (so for example in `application.properties`) using 'logging.level.*=LEVEL' where 'LEVEL' is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF. Example `application.properties`:

```
logging.level.org.springframework.web: DEBUG
logging.level.org.hibernate: ERROR
```

25.5 Custom log configuration

The various logging systems can be activated by including the appropriate libraries on the classpath, and further customized by providing a suitable configuration file in the root of the classpath, or in a location specified by the Spring `Environment` property `logging.config`. (Note however that since logging is initialized **before** the `ApplicationContext` is created, it isn't possible to control logging from `@PropertySources` in Spring `@Configuration` files. System properties and the conventional Spring Boot external configuration files work just fine.)

Depending on your logging system, the following files will be loaded:

Logging System	Customization
Logback	`logback.xml`
Log4j	`log4j.properties` or `log4j.xml`
Log4j2	`log4j2.xml`
JDK (Java Util Logging)	`logging.properties`

To help with the customization some other properties are transferred from the Spring `Environment` to System properties:

Spring Environment	System Property	Comments
`logging.file`	`LOG_FILE`	Used in default log configuration if defined.
`logging.path`	`LOG_PATH`	Used in default log configuration if defined.
`PID`	`PID`	The current process ID (discovered if possible and when not already defined as an OS environment variable).

All the logging systems supported can consult System properties when parsing their configuration files. See the default configurations in `spring-boot.jar` for examples.

> **Warning**
>
> There are know classloading issues with Java Util Logging that cause problems when running from an ‘executable jar’. We recommend that you avoid it if at all possible.

Spring Boot is well suited for web application development. You can easily create a self-contained HTTP server using embedded Tomcat, Jetty, or Undertow. Most web applications will use the `spring-boot-starter-web` module to get up and running quickly.

If you haven't yet developed a Spring Boot web application you can follow the "Hello World!" example in the *Getting started* section.

26.1 The 'Spring Web MVC framework'

The Spring Web MVC framework (often referred to as simply '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 using `@RequestMapping` annotations.

Here is a typical example `@RestController` to serve JSON data:

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

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

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

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

}
```

Spring MVC is part of the core Spring Framework and detailed information is available in the reference documentation. There are also several guides available at spring.io/guides that cover Spring MVC.

Spring MVC auto-configuration

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

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

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (see below).
- Automatic registration of `Converter`, `GenericConverter`, `Formatter` beans.
- Support for `HttpMessageConverters` (see below).
- Automatic registration of `MessageCodeResolver` (see below)
- Static `index.html` support.
- Custom `Favicon` support.

If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`. If you want to keep Spring Boot MVC features, and you just want to add additional MVC configuration (interceptors, formatters, view controllers etc.) you can add your own `@Bean` of type `WebMvcConfigurerAdapter`, but **without** `@EnableWebMvc`.

HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box, for example Objects can be automatically converted to JSON (using the Jackson library) or XML (using the Jackson XML extension if available, else using JAXB). Strings are encoded using `UTF-8` by default.

If you need to add or customize converters you can use Spring Boot's `HttpMessageConverters` class:

```
import org.springframework.boot.autoconfigure.web.HttpMessageConverters;
import org.springframework.context.annotation.*;
import org.springframework.http.converter.*;

@Configuration
public class MyConfiguration {

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

}
```

Any `HttpMessageConverter` bean that is present in the context will be added to the list of converters. You can also override default converters that way.

MessageCodesResolver

Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: `MessageCodesResolver`. Spring Boot will create one for you if you set the `spring.mvc.message-codes-resolver.format` property `PREFIX_ERROR_CODE` or `POSTFIX_ERROR_CODE` (see the enumeration in `DefaultMessageCodesResolver.Format`).

Static Content

By default Spring Boot will serve static content from a folder 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 you can modify that behavior by adding your own `WebMvcConfigurerAdapter` and overriding the `addResourceHandlers` method.

In a stand-alone web application the default servlet from the container is also enabled, and acts as a fallback, serving content from the root of the `ServletContext` if Spring decides not to handle it. Most of the time this will not happen (unless you modify the default MVC configuration) because Spring will always be able to handle requests through the `DispatcherServlet`.

In addition to the 'standard' static resource locations above, a special case is made for Webjars content. Any resources with a path in `/webjars/**` will be served from jar files if they are packaged in the Webjars format.

> **Tip**
>
> Do not use the `src/main/webapp` folder if your application will be packaged as a jar. Although this folder is a common standard, it will **only** work with war packaging and it will be silently ignored by most build tools if you generate a jar.

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 Velocity, FreeMarker and JSPs. Many other templating engines also ship their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Groovy
- Thymeleaf
- Velocity

When you're using one of these templating engines with the default configuration, your templates will be picked up automatically from `src/main/resources/templates`.

> **Tip**
>
> JSPs should be avoided if possible, there are several known limitations when using them with embedded servlet containers.

Error Handling

Spring Boot provides an `/error` mapping by default 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 will produce 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 just add a `View` that resolves to 'error'). To replace the default behaviour completely you can implement `ErrorController` and register a bean definition of that type, or simply add a bean of type `ErrorAttributes` to use the existing mechanism but replace the contents.

If you want more specific error pages for some conditions, the embedded servlet containers support a uniform Java DSL for customizing the error handling. For example:

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer(){
    return new MyCustomizer();
}

// ...

private static class MyCustomizer implements EmbeddedServletContainerCustomizer {

    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        container.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    }
```

```
}
```

You can also use regular Spring MVC features like `@ExceptionHandler` methods and `@ControllerAdvice`. The `ErrorController` will then pick up any unhandled exceptions.

N.B. if you register an `ErrorPage` with a path that will end up being handled by a `Filter` (e.g. 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, e.g.

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

(the default `FilterRegistrationBean` does not include the `ERROR` dispatcher type).

Error Handling on WebSphere Application Server

When deployed to a servlet container, a Spring Boot uses its error page filter to forward a request with an error status to the appropriate error page. The request can only be forwarded 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 behaviour by setting `com.ibm.ws.webcontainer.invokeFlushAfterService` to `false`

26.2 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 1.x and Apache Celtix work quite well out of the box if you just register their `Servlet` or `Filter` as a `@Bean` in your application context. Jersey 2.x 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 2.x just 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:

```
@Component
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        register(Endpoint.class);
    }

}
```

All the registered endpoints should be `@Components` with HTTP resource annotations (`@GET` etc.), e.g.

```
@Component
@Path("/hello")
public class Endpoint {

    @GET
    public String message() {
        return "Hello";
    }

}
```

Since the `Endpoint` is a Spring `@Component` its lifecycle is managed by Spring and you can `@Autowired` dependencies and inject external configuration with `@Value`. The Jersey servlet will be registered and mapped to `/*` by default. You can change the mapping by adding `@ApplicationPath` to your `ResourceConfig`.

By default Jersey will be set up as a Servlet in a `@Bean` of type `ServletRegistrationBean` named `jerseyServletRegistration`. 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 servlet has an `@Order` which you can set with `spring.jersey.filter.order`. Both the Servlet and the Filter registrations can be given init parameters using `spring.jersey.init.*` to specify a map of properties.

There is a Jersey sample so you can see how to set things up. There is also a Jersey 1.x sample. Note that in the Jersey 1.x sample that the spring-boot maven plugin has been configured to unpack some Jersey jars so they can be scanned by the JAX-RS implementation (because the sample asks for them to be scanned in its `Filter` registration). You may need to do the same if any of your JAX-RS resources are packages as nested jars.

26.3 Embedded servlet container support

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. Most developers will simply use the appropriate 'Starter POM' to obtain a fully configured instance. By default the embedded server will listen for HTTP requests on port `8080`.

Servlets and Filters

When using an embedded servlet container you can register Servlets and Filters directly as Spring beans. 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 will be mapped to `/`. In the case of multiple Servlet beans the bean name will be used as a path prefix. Filters will map to `/*`.

If convention-based mapping is not flexible enough you can use the `ServletRegistrationBean` and `FilterRegistrationBean` classes for complete control. You can also register items directly if your bean implements the `ServletContextInitializer` interface.

The EmbeddedWebApplicationContext

Under the hood Spring Boot uses a new type of `ApplicationContext` for embedded servlet container support. The `EmbeddedWebApplicationContext` is a special type of `WebApplicationContext` that bootstraps itself by searching for a single `EmbeddedServletContainerFactory` bean. Usually a `TomcatEmbeddedServletContainerFactory`, `JettyEmbeddedServletContainerFactory`, or `UndertowEmbeddedServletContainerFactory` will have been auto-configured.

> **Note**
>
> You usually won't need to be aware of these implementation classes. Most applications will be auto-configured and the appropriate `ApplicationContext` and `EmbeddedServletContainerFactory` will be created on your behalf.

Customizing embedded servlet containers

Common servlet container settings can be configured using Spring `Environment` properties. Usually you would define the properties in your `application.properties` file.

Common server settings include:

- `server.port` — The listen port for incoming HTTP requests.
- `server.address` — The interface address to bind to.
- `server.sessionTimeout` — A session timeout.

See the `ServerProperties` class for a complete list.

Programmatic customization

If you need to configure your embdedded servlet container programmatically you can register a Spring bean that implements the `EmbeddedServletContainerCustomizer` interface. `EmbeddedServletContainerCustomizer` provides access to the `ConfigurableEmbeddedServletContainer` which includes numerous customization setter methods.

```
import org.springframework.boot.context.embedded.*;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements EmbeddedServletContainerCustomizer {

    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        container.setPort(9000);
    }

}
```

Customizing ConfigurableEmbeddedServletContainer directly

If the above customization techniques are too limited, you can register the `TomcatEmbeddedServletContainerFactory`, `JettyEmbeddedServletContainerFactory` or `UndertowEmbeddedServletContainerFactory` bean yourself.

```
@Bean
public EmbeddedServletContainerFactory servletContainer() {
    TomcatEmbeddedServletContainerFactory factory = new TomcatEmbeddedServletContainerFactory();
    factory.setPort(9000);
    factory.setSessionTimeout(10, TimeUnit.MINUTES);
    factory.addErrorPages(new ErrorPage(HttpStatus.404, "/notfound.html");
    return factory;
}
```

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.

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 Tomcat it should work if you use war packaging, i.e. an executable war will work, and will also be deployable to a standard container (not limited to, but including Tomcat). An executable jar will not work because of a hard coded file pattern in Tomcat.
- Jetty does not currently work as an embedded container with JSPs.
- Undertow does not support JSPs.

There is a JSP sample so you can see how to set things up.

If Spring Security is on the classpath then web applications will be secure by default with 'basic' authentication on all HTTP endpoints. 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.

The default `AuthenticationManager` has a single user ('user' username and random password, printed at INFO level when the application starts up)

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

You can change the password by providing a `security.user.password`. This and other useful properties are externalized via `SecurityProperties` (properties prefix "security").

The default security configuration is implemented in `SecurityAutoConfiguration` and in the classes imported from there (`SpringBootWebSecurityConfiguration` for web security and `AuthenticationManagerConfiguration` for authentication configuration which is also relevant in non-web applications). To switch off the Boot default configuration completely in a web application you can add a bean with `@EnableWebSecurity`. To customize it you normally use external properties and beans of type `WebSecurityConfigurerAdapter` (e.g. to add form-based login). There are several secure applications in the Spring Boot samples to get you started with common use cases.

The basic features you get out of the box in a web application are:

- An `AuthenticationManager` bean with in-memory store and a single user (see `SecurityProperties.User` for the properties of the user).
- Ignored (unsecure) paths for common static resource locations (`/css/**`, `/js/**`, `/images/**` and `**/favicon.ico`).
- HTTP Basic security for all other endpoints.
- Security events published to Spring's `ApplicationEventPublisher` (successful and unsuccessful authentication and access denied).
- Common low-level features (HSTS, XSS, CSRF, caching) provided by Spring Security are on by default.

All of the above can be switched on and off or modified using external properties (`security.*`). To override the access rules without changing any other autoconfigured features add a `@Bean` of type `WebSecurityConfigurerAdapter` with `@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)`.

If the Actuator is also in use, you will find:

- The management endpoints are secure even if the application endpoints are unsecure.
- Security events are transformed into `AuditEvents` and published to the `AuditService`.
- The default user will have the `ADMIN` role as well as the `USER` role.

The Actuator security features can be modified using external properties (`management.security.*`). To override the application access rules add a `@Bean` of type `WebSecurityConfigurerAdapter` and use `@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)` if you *don't* want to override the

actuator access rules, or `@Order(ManagementServerProperties.ACCESS_OVERRIDE_ORDER)` if you *do* want to override the actuator access rules.

The Spring Framework provides extensive support for working with SQL databases. From direct JDBC access using `JdbcTemplate` to complete 'object relational mapping' technologies such as Hibernate. Spring Data provides an additional level of functionality, creating `Repository` implementations directly from interfaces and using conventions to generate queries from your method names.

28.1 Configure a DataSource

Java's `javax.sql.DataSource` interface provides a standard method of working with database connections. Traditionally a DataSource uses a `URL` along with some credentials to establish a database connection.

Embedded Database Support

It's often convenient to develop applications using an in-memory embedded database. Obviously, in-memory databases do not provide persistent storage; you will need to populate your database when your application starts and be prepared to throw away data when your application ends.

> **Tip**
>
> The 'How-to' section includes a *section on how to initialize a database*

Spring Boot can auto-configure embedded H2, HSQL and Derby databases. You don't need to provide any connection URLs, simply include a build dependency to the embedded database that you want to use.

For example, typical POM dependencies would be:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>runtime</scope>
</dependency>
```

> **Note**
>
> You need a dependency on `spring-jdbc` for an embedded database to be auto-configured. In this example it's pulled in transitively via `spring-boot-starter-data-jpa`.

Connection to a production database

Production database connections can also be auto-configured using a pooling `DataSource`. Here's the algorithm for choosing a specific implementation:

- We prefer the Tomcat pooling `DataSource` for its performance and concurrency, so if that is available we always choose it.
- If HikariCP is available we will use it
- If Commons DBCP is available we will use it, but we don't recommend it in production.

- Lastly, if Commons DBCP2 is available we will use it

If you use the `spring-boot-starter-jdbc` or `spring-boot-starter-data-jpa` 'starter POMs' you will automcatically get a dependency to `tomcat-jdbc`.

> **Note**
>
> Additional connection pools can always be configured manually. If you define your own `DataSource` bean, auto-configuration will not occur.

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

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

See `DataSourceProperties` for more of the supported options.

> **Tip**
>
> You often won't need to specify the `driver-class-name` since Spring boot can deduce it for most databases from the `url`.

> **Note**
>
> For a pooling `DataSource` to be created we need to be able to verify that a valid `Driver` class is available, so we check for that before doing anything. I.e. if you set `spring.datasource.driverClassName=com.mysql.jdbc.Driver` then that class has to be loadable.

Connection to a JNDI DataSource

If you are deploying your Spring Boot application to an Application Server you might want to configure and manage your DataSource using your Application Servers built-in features and access it using JNDI.

The `spring.datasource.jndi-name` property can be used as an alternative to the `spring.datasource.url`, `spring.datasource.username` and `spring.datasource.password` properties to access the `DataSource` from a specific JNDI location. For example, the following section in `application.properties` shows how you can access a JBoss AS defined `DataSource`:

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

28.2 Using JdbcTemplate

Spring's `JdbcTemplate` and `NamedParameterJdbcTemplate` classes are auto-configured and you can `@Autowire` them directly into your own beans:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;
```

```
@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

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

    // ...

}
```

28.3 JPA and 'Spring Data'

The Java Persistence API is a standard technology that allows you to 'map' objects to relational databases. The `spring-boot-starter-data-jpa` POM provides a quick way to get started. It provides the following key dependencies:

- Hibernate — One of the most popular JPA implementations.
- Spring Data JPA — Makes it easy to easily implement JPA-based repositories.
- Spring ORMs — Core ORM support from the Spring Framework.

> **Tip**
>
> We won't go into too many details of JPA or Spring Data here. You can follow the 'Accessing Data with JPA' guide from spring.io and read the Spring Data JPA and Hibernate reference documentation.

Entity Classes

Traditionally, JPA 'Entity' classes are specified in a `persistence.xml` file. With Spring Boot this file is not necessary and instead 'Entity Scanning' is used. By default all packages below your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) will be searched.

Any classes annotated with `@Entity`, `@Embeddable` or `@MappedSuperclass` will be considered. A typical entity class would look something like this:

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

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

@Entity
public class City implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String state;
```

```
    // ... additional members, often include @OneToMany mappings

    protected City() {
        // no-args constructor required by JPA spec
        // this one is protected since it shouldn't be used directly
    }

    public City(String name, String state) {
        this.name = name;
        this.country = country;
    }

    public String getName() {
        return this.name;
    }

    public String getState() {
        return this.state;
    }

    // ... etc

}
```

Tip

You can customize entity scanning locations using the `@EntityScan` annotation. See the *Section 67.4, "Separate @Entity definitions from Spring configuration"* how-to.

Spring Data JPA Repositories

Spring Data JPA repositories are interfaces that you can define to access data. JPA queries are created automatically from your method names. For example, a `CityRepository` interface might declare a `findAllByState(String state)` method to find all cities in a given state.

For more complex queries you can annotate your method using Spring Data's `Query` annotation.

Spring Data repositories usually extend from the `Repository` or `CrudRepository` interfaces. If you are using auto-configuration, repositories will be searched from the package containing your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) down.

Here is a typical Spring Data repository:

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

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

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

Tip

We have barely scratched the surface of Spring Data JPA. For complete details check their reference documentation.

Creating and dropping JPA databases

By default, JPA databases will be automatically created **only** if you use an embedded database (H2, HSQL or Derby). You can explicitly configure JPA settings using `spring.jpa.*` properties. For example, to create and drop tables you can add the following to your `application.properties`.

```
spring.jpa.hibernate.ddl-auto=create-drop
```

> **Note**
>
> Hibernate's own internal property name for this (if you happen to remember it better) is `hibernate.hbm2ddl.auto`. You can set it, along with other Hibernate native properties, using `spring.jpa.properties.*` (the prefix is stripped before adding them to the entity manager). Example:

```
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```

passes `hibernate.globally_quoted_identifiers` to the Hibernate entity manager.

By default the DDL execution (or validation) is deferred until the `ApplicationContext` has started. There is also a `spring.jpa.generate-ddl` flag, but it is not used if Hibernate autoconfig is active because the `ddl-auto` settings are more fine-grained.

Spring Data provides additional projects that help you access a variety of NoSQL technologies including MongoDB, Neo4J, Elasticsearch, Solr, Redis, Gemfire, Couchbase and Cassandra. Spring Boot provides auto-configuration for Redis, MongoDB, Elasticsearch, Solr and Gemfire; you can make use of the other projects, but you will need to configure them yourself. Refer to the appropriate reference documentation at projects.spring.io/spring-data.

29.1 Redis

Redis is a cache, message broker and richly-featured key-value store. Spring Boot offers basic auto-configuration for the Jedis client library and abstractions on top of it provided by Spring Data Redis. There is a `spring-boot-starter-redis` 'Starter POM' for collecting the dependencies in a convenient way.

Connecting to Redis

You can inject an auto-configured `RedisConnectionFactory`, `StringRedisTemplate` or vanilla `RedisTemplate` instance as you would any other Spring Bean. By default the instance will attempt to connect to a Redis server using `localhost:6379`:

```
@Component
public class MyBean {

    private StringRedisTemplate template;

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

    // ...

}
```

If you add a `@Bean` of your own of any of the auto-configured types it will replace the default (except in the case of `RedisTemplate` the exclusion is based on the bean name 'redisTemplate' not its type). If `commons-pool2` is on the classpath you will get a pooled connection factory by default.

29.2 MongoDB

MongoDB is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several conveniences for working with MongoDB, including the The `spring-boot-starter-data-mongodb` 'Starter POM'.

Connecting to a MongoDB database

You can inject an auto-configured `org.springframework.data.mongodb.MongoDbFactory` to access Mono databases. By default the instance will attempt to connect to a MongoDB server using the URL `mongodb://localhost/test`:

```
import org.springframework.data.mongodb.MongoDbFactory;
import import com.mongodb.DB;

@Component
public class MyBean {

    private final MongoDbFactory mongo;
```

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

    // ...

    public void example() {
        DB db = mongo.getDb();
        // ...
    }

}
```

You can set `spring.data.mongodb.uri` property to change the `url`, or alternatively specify a `host`/`port`. For example, you might declare the following in your `application.properties`:

```
spring.data.mongodb.host=mongoserver
spring.data.mongodb.port=27017
```

> **Tip**
>
> If `spring.data.mongodb.port` is not specified the default of `27017` is used. You could simply delete this line from the sample above.

> **Tip**
>
> If you aren't using Spring Data Mongo you can inject `com.mongodb.Mongo` beans instead of using `MongoDbFactory`.

You can also declare your own `MongoDbFactory` or `Mongo` `@Beans` if you want to take complete control of establishing the MongoDB connection.

MongoTemplate

Spring Data Mongo provides a `MongoTemplate` class that is very similar in its design to Spring's `JdbcTemplate`. As with `JdbcTemplate` Spring Boot auto-configures a bean for you to simply inject:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final MongoTemplate mongoTemplate;

    @Autowired
    public MyBean(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    // ...

}
```

See the `MongoOperations` Javadoc for complete details.

Spring Data MongoDB repositories

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

In fact, both Spring Data JPA and Spring Data MongoDB share the same common infrastructure; so you could take the JPA example from earlier and, assuming that `City` is now a Mongo data class rather than a JPA `@Entity`, it will work in the same way.

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

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

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

> **Tip**
>
> For complete details of Spring Data MongoDB, including its rich object mapping technologies, refer to their reference documentation.

29.3 Gemfire

Spring Data Gemfire provides convenient Spring-friendly tools for accessing the Pivotal Gemfire data management platform. There is a `spring-boot-starter-data-gemfire` 'Starter POM' for collecting the dependencies in a convenient way. There is currently no auto=config support for Gemfire, but you can enable Spring Data Repositories with a single annotation.

29.4 Solr

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

Connecting to Solr

You can inject an auto-configured `SolrServer` instance as you would any other Spring Bean. By default the instance will attempt to connect to a server using `localhost:8983/solr`:

```
@Component
public class MyBean {

    private SolrServer solr;

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

    // ...

}
```

If you add a `@Bean` of your own of type `SolrServer` it will replace the default.

Spring Data Solr repositories

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

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

> **Tip**
>
> For complete details of Spring Data Solr, refer to their reference documentation.

29.5 Elasticsearch

Elastic Search is an open source, distributed, real-time search and analytics engine. Spring Boot offers basic auto-configuration for the Elasticsearch and abstractions on top of it provided by Spring Data Elasticsearch. There is a `spring-boot-starter-data-elasticsearch` 'Starter POM' for collecting the dependencies in a convenient way.

Connecting to Elasticsearch

You can inject an auto-configured `ElasticsearchTemplate` or Elasticsearch `Client` instance as you would any other Spring Bean. By default the instance will attempt to connect to a local in-memory server (a `NodeClient` in Elasticsearch terms), but you can switch to a remote server (i.e. a `TransportClient`) by setting `spring.data.elasticsearch.clusterNodes` to a comma-separated 'host:port' list.

```
@Component
public class MyBean {

    private ElasticsearchTemplate template;

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

    // ...

}
```

If you add a `@Bean` of your own of type `ElasticsearchTemplate` it will replace the default.

Spring Data Elasticsearch repositories

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

In fact, both Spring Data JPA and Spring Data Elasticsearch share the same common infrastructure; so you could take the JPA example from earlier and, assuming that `City` is now an Elasticsearch `@Document` class rather than a JPA `@Entity`, it will work in the same way.

> **Tip**
>
> For complete details of Spring Data Elasticsearch, refer to their reference documentation.

The Spring Framework provides extensive support for integrating with messaging systems: from simplified use of the JMS API using `JmsTemplate` to a complete infrastructure to receive messages asynchronously. Spring AMQP provides a similar feature set for the 'Advanced Message Queuing Protocol' and Boot also provides auto-configuration options for `RabbitTemplate` and RabbitMQ. There is also support for STOMP messaging natively in Spring Websocket and Spring Boot has support for that through starters and a small amount of auto-configuration.

30.1 JMS

The `javax.jms.ConnectionFactory` interface provides a standard method of creating a `javax.jms.Connection` for interacting with a JMS broker. Although Spring needs a `ConnectionFactory` to work with JMS, you generally won't need to use it directly yourself and you can instead rely on higher level messaging abstractions (see the relevant section of the Spring Framework reference documentation for details). Spring Boot also auto configures the necessary infrastructure to send and receive messages.

HornetQ support

Spring Boot can auto-configure a `ConnectionFactory` when it detects that HornetQ is available on the classpath. If the broker is present, an embedded broker is started and configured automatically (unless the mode property has been explicitly set). The supported modes are: `embedded` (to make explicit that an embedded broker is required and should lead to an error if the broker is not available in the classpath), and `native` to connect to a broker using the the `netty` transport protocol. When the latter is configured, Spring Boot configures a `ConnectionFactory` connecting to a broker running on the local machine with the default settings.

> **Note**
>
> If you are using `spring-boot-starter-hornetq` the necessary dependencies to connect to an existing HornetQ instance are provided, as well as the Spring infrastructure to integrate with JMS. Adding `org.hornetq:hornetq-jms-server` to your application allows you to use the embedded mode.

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

```
spring.hornetq.mode=native
spring.hornetq.host=192.168.1.210
spring.hornetq.port=9876
```

When embedding the broker, you can chose if you want to enable persistence, and the list of destinations that should be made available. These can be specified as a comma-separated list to create them with the default options; or you can define bean(s) of type `org.hornetq.jms.server.config.JMSQueueConfiguration` or `org.hornetq.jms.server.config.TopicConfiguration`, for advanced queue and topic configurations respectively.

See `HornetQProperties` for more of the supported options.

No JNDI lookup is involved at all and destinations are resolved against their names, either using the 'name' attribute in the HornetQ configuration or the names provided through configuration.

ActiveMQ support

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

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

```
spring.activemq.broker-url=tcp://192.168.1.210:9876
spring.activemq.user=admin
spring.activemq.password=secret
```

See `ActiveMQProperties` for more of the supported options.

By default, ActiveMQ creates a destination if it does not exist yet, so destinations are resolved against their provided names.

Using a JNDI ConnectionFactory

If you are running your application in an Application Server Spring Boot will attempt to locate a JMS `ConnectionFactory` using JNDI. By default the locations `java:/JmsXA` and `java:/XAConnectionFactory` will be checked. You can use the `spring.jms.jndi-name` property if you need to specify an alternative location:

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

Sending a message

Spring's `JmsTemplate` is auto-configured and you can autowire it directly into your own beans:

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

@Component
public class MyBean {

    private final JmsTemplate jmsTemplate;

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

    // ...

}
```

> **Note**
>
> `JmsMessagingTemplate` (new in Spring 4.1) can be injected in a similar manner.

Receiving a message

When the JMS infrastructure is present, any bean can be annotated with `@JmsListener` to create a listener endpoint. If no `JmsListenerContainerFactory` has been defined, a default one is configured automatically.

The following component creates a listener endpoint on the `someQueue` destination:

```
@Component
public class MyBean {

    @JmsListener(destination = "someQueue")
    public void processMessage(String content) {
        // ...
    }

}
```

Check the javadoc of `@EnableJms` for more details.

The Spring Framework provides an easy abstraction for sending email using the `JavaMailSender` interface and Spring Boot provides auto-configuration for it as well as a starter module.

> **Tip**
>
> Check the reference documentation for a detailed explanation of how you can use `JavaMailSender`.

If `spring.mail.host` and the relevant libraries (as defined by `spring-boot-starter-mail`) are available, a default `JavaMailSender` is created if none exists. The sender can be further customized by configuration items from the `spring.mail` namespace, see the `MailProperties` for more details.

Spring Boot supports distributed JTA transactions across multiple XA resources using either an Atomkos or Bitronix embedded transaction manager. JTA transactions are also supported when deploying to a suitable Java EE Application Server.

When a JTA environment is detected, Spring's `JtaTransactionManager` will be used to manage transactions. Auto-configured JMS, DataSource and JPA beans will be upgraded to support XA transactions. You can use standard Spring idioms such as `@Transactional` to participate in a distributed transaction. If you are within a JTA environment and still want to use local transactions you can set the `spring.jta.enabled` property to `false` to disable the JTA auto-configuration.

32.1 Using an Atomikos transaction manager

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

By default Atomikos transaction logs will be written to a `transaction-logs` folder in your application home directory (the directory in which your application jar file resides). You can customize this directory by setting a `spring.jta.log-dir` property in your `application.properties` file. Properties starting `spring.jta.` can also be used to customize the Atomikos `UserTransactionServiceIml`. See the `AtomikosProperties` javadoc for complete details.

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

32.2 Using a Bitronix transaction manager

Bitronix is another popular open source JTA transaction manager implementation. You can use the `spring-boot-starter-jta-bitronix` starter POM to add the appropriate Birtronix dependencies to your project. As with Atomikos, Spring Boot will automatically configure Bitronix and post-process your beans to ensure that startup and shutdown ordering is correct.

By default Bitronix transaction log files (`part1.btm` and `part2.btm`) will be written to a `transaction-logs` folder in your application home directory. You can customize this directory by using the `spring.jta.log-dir` property. Properties starting `spring.jta.` are also bound to the `bitronix.tm.Configuration` bean, allowing for complete customization. See the Bitronix documentation for details.

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

32.3 Using a Java EE managed transaction manager

If you are packaging your Spring Boot application as a `war` or `ear` file and deploying it to a Java EE application server, you can use your application servers built-in transaction manager. Spring Boot will attempt to auto-configure a transaction manager by looking at common JNDI locations (`java:comp/UserTransaction`, `java:comp/TransactionManager` etc). If you are using a transaction service provided by your application server, you will generally also want to ensure that all resources are managed by the server and exposed over JNDI. Spring Boot will attempt to auto-configure JMS by looking for a `ConnectionFactory` at the JNDI path `java:/JmsXA` or `java:/XAConnectionFactory` and you can use the `spring.datasource.jndi-name` property to configure your `DataSource`.

32.4 Mixing XA and non-XA JMS connections

When using JTA, the primary JMS `ConnectionFactory` bean will be XA aware and participate in distributed transactions. In some situations you might want to process certain JMS messages using a non-XA `ConnectionFactory`. For example, your JMS processing logic might take longer than the XA timeout.

If you want to use a non-XA `ConnectionFactory` you can inject the `nonXaJmsConnectionFactory` bean rather than the `@Primary` `jmsConnectionFactory` bean. For consistency the `jmsConnectionFactory` bean is also provided using the bean alias `xaJmsConnectionFactory`.

For example:

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

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

// Inject the non-XA aware ConnectionFactory
@Autowired
@Qualifier("nonXaJmsConnectionFactory")
private ConnectionFactory nonXaConnectionFactory;
```

32.5 Supporting an alternative embedded transaction manager

The `XAConnectionFactoryWrapper` and `XADataSourceWrapper` interfaces can be used to support alternative embedded transaction managers. The interfaces are responsible for wrapping `XAConnectionFactory` and `XADataSource` beans and exposing them as regular `ConnectionFactory` and `DataSource` beans which will transparently enroll in the distributed transaction. DataSource and JMS auto-configuration will use JTA variants as long as you have a `JtaTransactionManager` bean and appropriate XA wrapper beans registered within your `ApplicationContext`

The BitronixXAConnectionFactoryWrapper and BitronixXADataSourceWrapper provide good examples of how to write XA wrappers.

Spring Integration provides abstractions over messaging and also other transports such as HTTP, TCP etc. If Spring Integration is available on your classpath it will be initialized through the `@EnableIntegration` annotation. Message processing statistics will be published over JMX if `'spring-integration-jmx'` is also on the classpath. See the `IntegrationAutoConfiguration` class for more details.

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. By default Spring Boot will create an `MBeanServer` with bean id 'mbeanServer' and expose any of your beans that are annotated with Spring JMX annotations (`@ManagedResource`, `@ManagedAttribute`, `@ManagedOperation`).

See the `JmxAutoConfiguration` class for more details.

Spring Boot provides a number of useful tools for testing your application. The `spring-boot-starter-test` POM provides Spring Test, JUnit, Hamcrest and Mockito dependencies. There are also useful test utilities in the core `spring-boot` module under the `org.springframework.boot.test` package.

35.1 Test scope dependencies

If you use the `spring-boot-starter-test` 'Starter POM' (in the `test scope`), you will find the following provided libraries:

- Spring Test — integration test support for Spring applications.
- JUnit — The de-facto standard for unit testing Java applications.
- Hamcrest — A library of matcher objects (also known as constraints or predicates) allowing `assertThat` style JUnit assertions.
- Mockito — A Java mocking framework.

These are common libraries that we generally find useful when writing tests. You are free to add additional test dependencies of your own if these don't suit your needs.

35.2 Testing Spring applications

One of the major advantages of dependency injection is that it should make your code easier to unit test. You can simply instantiate objects 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` actually involved in the process). It's 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 just such integration testing. You can declare a dependency directly to `org.springframework:spring-test` or use the `spring-boot-starter-test` 'Starter POM' 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.

35.3 Testing Spring Boot applications

A Spring Boot application is just 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. One thing to watch out for though is that the external properties, logging and other features of Spring Boot are only installed in the context by default if you use `SpringApplication` to create it.

Spring Boot provides a `@SpringApplicationConfiguration` annotation as an alternative to the standard `spring-test` `@ContextConfiguration` annotation. If you use `@SpringApplicationConfiguration` to configure the `ApplicationContext` used in your tests, it will be created via `SpringApplication` and you will get the additional Spring Boot features.

For example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SampleDataJpaApplication.class)
public class CityRepositoryIntegrationTests {

    @Autowired
    CityRepository repository;

    // ...

}
```

Tip

The context loader guesses whether you want to test a web application or not (e.g. with `MockMVC`) by looking for the `@WebAppConfiguration` annotation. (`MockMVC` and `@WebAppConfiguration` are part of `spring-test`).

If you want a web application to start up and listen on its normal port, so you can test it with HTTP (e.g. using `RestTemplate`), annotate your test class (or one of its superclasses) with `@IntegrationTest`. This can be very useful because it means you can test the full stack of your application, but also inject its components into the test class and use them to assert the internal state of the application after an HTTP interaction. For example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SampleDataJpaApplication.class)
@WebAppConfiguration
@IntegrationTest
public class CityRepositoryIntegrationTests {

    @Autowired
    CityRepository repository;

    RestTemplate restTemplate = new TestRestTemplate();

    // ... interact with the running server

}
```

Note

Spring's test framework will cache application contexts between tests. Therefore, as long as your tests share the same configuration, the time consuming process of starting and stopping the server will only happen once, regardless of the number of tests that actually run.

To change the port you can add environment properties to `@IntegrationTest` as colon- or equals-separated name-value pairs, e.g. `@IntegrationTest("server.port:9000")`. Additionally you can set the `server.port` and `management.port` properties to `0` in order to run your integration tests using random ports. For example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MyApplication.class)
@WebAppConfiguration
@IntegrationTest({"server.port=0", "management.port=0"})
public class SomeIntegrationTests {

    // ...

}
```

See Section 64.4, “Discover the HTTP port at runtime” for a description of how you can discover the actual port that was allocated for the duration of the tests.

Using Spock to test Spring Boot applications

If you wish to use Spock to test a Spring Boot application you should add a dependency on Spock’s `spock-spring` module to your application’s build. `spock-spring` integrates Spring’s test framework into Spock.

Please note that you cannot use the `@SpringApplicationConfiguration` annotation that was described above as Spock does not find the `@ContextConfiguration` meta-annotation. To work around this limitation, you should use the `@ContextConfiguration` annotation directly and configure it to use the Spring Boot specific context loader:

```
@ContextConfiguration(loader = SpringApplicationContextLoader.class)
class ExampleSpec extends Specification {

    // ...

}
```

Note

The annotations described above can be used with Spock, i.e. you can annotate your `Specification` with `@IntegrationTest` and `@WebAppConfiguration` to suit the needs of your tests.

35.4 Test utilities

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

ConfigFileApplicationContextInitializer

`ConfigFileApplicationContextInitializer` is an `ApplicationContextInitializer` that can apply to your tests to load Spring Boot `application.properties` files. You can use this when you don’t need the full features provided by `@SpringApplicationConfiguration`.

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

EnvironmentTestUtils

`EnvironmentTestUtils` allows you to quickly add properties to a `ConfigurableEnvironment` or `ConfigurableApplicationContext`. Simply call it with `key=value` strings:

```
EnvironmentTestUtils.addEnvironment(env, "org=Spring", "name=Boot");
```

OutputCapture

`OutputCapture` is a JUnit `Rule` that you can use to capture `System.out` and `System.err` output. Simply declare the capture as a `@Rule` then use `toString()` for assertions:

```
import org.junit.Rule;
import org.junit.Test;
```

```
import org.springframework.boot.test.OutputCapture;

import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

public class MyTest {

 @Rule
 public OutputCapture capture = new OutputCapture();

 @Test
 public void testName() throws Exception {
  System.out.println("Hello World!");
  assertThat(capture.toString(), containsString("World"));
 }

}
```

TestRestTemplate

`TestRestTemplate` is a convenience subclass of 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 will behave in a test-friendly way: not following redirects (so you can assert the response location), ignoring cookies (so the template is stateless), and not throwing exceptions on server-side errors. It is recommended, but not mandatory, to use Apache HTTP Client (version 4.3.2 or better), and if you have that on your classpath the `TestRestTemplate` will respond by configuring the client appropriately.

```
public class MyTest {

 RestTemplate template = new TestRestTemplate();

 @Test
 public void testRequest() throws Exception {
  HttpHeaders headers = template.getForEntity("http://myhost.com", String.class).getHeaders();
  assertThat(headers.getLocation().toString(), containsString("myotherhost"));
 }

}
```

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.

36.1 Understanding auto-configured beans

Under the hood, auto-configuration is implemented with standard `@Configuration` classes. Additional `@Conditional` annotations are used to constrain when the auto-configuration should apply. Usually auto-configuration classes use `@ConditionalOnClass` and `@ConditionalOnMissingBean` annotations. This ensures that auto-configuration only applies when relevant classes are found and when you have not declared your own `@Configuration`.

You can browse the source code of `spring-boot-autoconfigure` to see the `@Configuration` classes that we provide (see the `META-INF/spring.factories` file).

36.2 Locating auto-configuration candidates

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

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

You can use the `@AutoConfigureAfter` or `@AutoConfigureBefore` annotations if your configuration needs to be applied in a specific order. For example, if you provide web-specific configuration, your class may need to be applied after `WebMvcAutoConfiguration`.

36.3 Condition annotations

You almost always want to include one or more `@Condition` annotations on your auto-configuration class. The `@ConditionalOnMissingBean` 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.

Class conditions

The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations allows configuration to be skipped based on the presence or absence of specific classes. Due to the fact that annotation metadata is parsed using ASM you can actually 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 using a `String` value.

Bean conditions

The `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations allow configurations to be skipped 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 allows you to limit the `ApplicationContext` hierarchy that should be considered when searching for beans.

> **Note**
>
> `@Conditional` annotations are processed when `@Configuration` classes are parsed. Auto-configure `@Configuration` is always parsed last (after any user defined beans), however, if you are using these annotations on regular `@Configuration` classes, care must be taken not to refer to bean definitions that have not yet been created.

Property conditions

The `@ConditionalOnProperty` annotation allows configuration to 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` will be matched. You can also create more advanced checks using the `havingValue` and `matchIfMissing` attributes.

Resource conditions

The `@ConditionalOnResource` annotation allows configuration to be included only when a specific resource is present. Resources can be specified using the usual Spring conventions, for example, `file:/home/user/test.dat`.

Web Application Conditions

The `@ConditionalOnWebApplication` and `@ConditionalOnNotWebApplication` annotations allow configuration to be skipped depending on whether the application is a 'web application'. A web application is any application that is using a Spring `WebApplicationContext`, defines a `session` scope or has a `StandardServletEnvironment`.

SpEL expression conditions

The `@ConditionalOnExpression` annotation allows configuration to be skipped based on the result of a SpEL expression.

Spring Boot provides WebSockets auto-configuration for embedded Tomcat (8 and 7), Jetty 9 and Undertow. If you're deploying a war file to a standalone container, Spring Boot assumes that the container will be responsible for the configuration of its WebSocket support.

Spring Framework provides rich WebSocket support that can be easily accessed via the `spring-boot-starter-websocket` module.

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

If you are comfortable with Spring Boot's core features, you can carry on and read about production-ready features.

Part V. Spring Boot Actuator: Production-ready features

Spring Boot includes a number of additional features to help you monitor and manage your application when it’s pushed to production. You can choose to manage and monitor your application using HTTP endpoints, with JMX or even by remote shell (SSH or Telnet). Auditing, health and metrics gathering can be automatically applied to your application.

The `spring-boot-actuator` module provides all of Spring Boot's production-ready features. The simplest way to enable the features is to add a dependency to the `spring-boot-starter-actuator` 'Starter POM'.

> **Definition of Actuator**
>
> An actuator is a manufacturing term, referring to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change.

To add the actuator to a Maven based project, add the following 'starter' dependency:

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

For Gradle, use the declaration:

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-actuator")
}
```

Actuator endpoints allow you to monitor and interact with your application. Spring Boot includes a number of built-in endpoints and you can also add your own. For example the `health` endpoint provides basic application health information.

The way that endpoints are exposed will depend on the type of technology that you choose. Most applications choose HTTP monitoring, where the ID of the endpoint is mapped to a URL. For example, by default, the `health` endpoint will be mapped to `/health`.

The following endpoints are available:

ID	Description	Sensitive
`autoconfig`	Displays an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied.	true
`beans`	Displays a complete list of all the Spring Beans in your application.	true
`configprops`	Displays a collated list of all `@ConfigurationProperties`.	true
`dump`	Performs a thread dump.	true
`env`	Exposes properties from Spring's `ConfigurableEnvironment`.	true
`health`	Shows application health information (a simple 'status' when accessed over an unauthenticated connection or full message details when authenticated).	false
`info`	Displays arbitrary application info.	false
`metrics`	Shows 'metrics' information for the current application.	true
`mappings`	Displays a collated list of all `@RequestMapping` paths.	true
`shutdown`	Allows the application to be gracefully shutdown (not enabled by default).	true
`trace`	Displays trace information (by default the last few HTTP requests).	true

> **Note**
>
> Depending on how an endpoint is exposed, the `sensitive` parameter may be used as a security hint. For example, sensitive endpoints will require a username/password when they are accessed over HTTP (or simply disabled if web security is not enabled).

40.1 Customizing endpoints

Endpoints can be customized using Spring properties. You can change if an endpoint is `enabled`, if it is considered `sensitive` and even its `id`.

For example, here is an `application.properties` that changes the sensitivity and id of the `beans` endpoint and also enables `shutdown`.

```
endpoints.beans.id=springbeans
endpoints.beans.sensitive=false
endpoints.shutdown.enabled=true
```

Note

The prefix `#endpoints + . + name`" is used to uniquely identify the endpoint that is being configured.

By default, all endpoints except for `shutdown` are enabled. If you prefer to specifically "opt-in" endpoint enablement you can use the `endpoints.enabled` property. For example, the following will disable *all* endpoints except for `info`:

```
endpoints.enabled=false
endpoints.info.enabled=true
```

40.2 Health information

Health information can be used to check the status of your running application. It is often used by monitoring software to alert someone if a production system goes down. The default information exposed by the `health` endpoint depends on how it is accessed. For an insecure unauthenticated connection a simple 'status' message is returned, for a secure or authenticated connection additional details are also displayed (see Section 41.6, "HTTP Health endpoint access restrictions" for HTTP details).

Health information is collected from all `HealthIndicator` beans defined in your `ApplicationContext`. Spring Boot includes a number of auto-configured `HealthIndicators` and you can also write your own.

40.3 Security with HealthIndicators

Information returned by `HealthIndicators` is often somewhat sensitive in nature. For example, you probably don't want to publish details of your database server to the world. For this reason, by default, only the health status is exposed over an unauthenticated HTTP connection. If you are happy for complete health information to always be exposed you can set `endpoints.health.sensitive` to `false`.

Health responses are also cached to prevent "denial of service" attacks. Use the `endpoints.health.time-to-live` property if you want to change the default cache period of 1000 milliseconds.

Auto-configured HealthIndicators

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

Name	Description
`DiskSpaceHealthIndicator`	Checks for low disk space.
`DataSourceHealthIndicator`	Checks that a connection to `DataSource` can be obtained.
`MongoHealthIndicator`	Checks that a Mongo database is up.
`RabbitHealthIndicator`	Checks that a Rabbit server is up.
`RedisHealthIndicator`	Checks that a Redis server is up.

Name	Description
SolrHealthIndicator	Checks that a Solr server is up.

Writing custom HealthIndicators

To provide custom health information you can register Spring beans that implement the `HealthIndicator` interface. You need to provide an implementation of the `health()` method and return a `Health` response. The `Health` response should include a status and can optionally include additional details to be displayed.

```
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealth implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode);
        }
        return Health.up();
    }

}
```

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

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

```
management.health.status.order: DOWN, OUT_OF_SERVICE, UNKNOWN, UP
```

You might also want to register custom status mappings with the `HealthMvcEndpoint` if you access the health endpoint over HTTP. For example you could map `FATAL` to `HttpStatus.SERVICE_UNAVAILABLE`.

40.4 Custom application info information

You can customize the data exposed by the `info` endpoint by setting `info.*` Spring properties. All `Environment` properties under the info key will be automatically exposed. For example, you could add the following to your `application.properties`:

```
info.app.name=MyService
info.app.description=My awesome service
info.app.version=1.0.0
```

Automatically expand info properties at build time

Rather than hardcoding some properties that are also specified in your project's build configuration, you can automatically expand info properties using the existing build configuration instead. This is possible in both Maven and Gradle.

Automatic property expansion using Maven

You can automatically expand info properties from the Maven project using resource filtering. If you use the `spring-boot-starter-parent` you can then refer to your Maven 'project properties' via `@..@` placeholders, e.g.

```
project.artifactId=myproject
project.name=Demo
project.version=X.X.X.X
project.description=Demo project for info endpoint
info.build.artifact=@project.artifactId@
info.build.name=@project.name@
info.build.description=@project.description@
info.build.version=@project.version@
```

> **Note**
>
> In the above example we used `project.*` to set some values to be used as fallbacks if the Maven resource filtering has not been switched on for some reason.

> **Note**
>
> If you don't use the starter parent, in your `pom.xml` you need (inside the `<build/>` element):

```
<resources>
    <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
    </resource>
</resources>
```

and (inside `<plugins/>`):

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-resources-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <delimiters>
            <delimiter>@</delimiter>
        </delimiters>
    </configuration>
</plugin>
```

Automatic property expansion using Gradle

You can automatically expand info properties from the Gradle project by configuring the Java plugin's `processResources` task to do so:

```
processResources {
    expand(project.properties)
}
```

You can then refer to your Gradle project's properties via placeholders, e.g.

```
info.build.name=${name}
info.build.description=${description}
info.build.version=${version}
```

Git commit information

Another useful feature of the `info` endpoint is its ability to publish information about the state of your `git` source code repository when the project was built. If a `git.properties` file is contained in your jar the `git.branch` and `git.commit` properties will be loaded.

For Maven users the `spring-boot-starter-parent` POM includes a pre-configured plugin to generate a `git.properties` file. Simply add the following declaration to your POM:

```
<build>
    <plugins>
        <plugin>
            <groupId>pl.project13.maven</groupId>
            <artifactId>git-commit-id-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

A similar `gradle-git` plugin is also available for Gradle users, although a little more work is required to generate the properties file.

If you are developing a Spring MVC application, Spring Boot Actuator will auto-configure all enabled endpoints to be exposed over HTTP. The default convention is to use the `id` of the endpoint as the URL path. For example, `health` is exposed as `/health`.

41.1 Securing sensitive endpoints

If you add 'Spring Security' to your project, all sensitive endpoints exposed over HTTP will be protected. By default 'basic' authentication will be used with the username `user` and a generated password (which is printed on the console when the application starts).

> **Tip**
>
> Generated passwords are logged as the application starts. Search for 'Using default security password'.

You can use Spring properties to change the username and password and to change the security role required to access the endpoints. For example, you might set the following in your `application.properties`:

```
security.user.name=admin
security.user.password=secret
management.security.role=SUPERUSER
```

> **Tip**
>
> If you don't use Spring Security and your HTTP endpoints are exposed publicly, you should carefully consider which endpoints you enable. See Section 40.1, "Customizing endpoints" for details of how you can set `endpoints.enabled` to `false` then "opt-in" only specific endpoints.

41.2 Customizing the management server context path

Sometimes it is useful to group all management endpoints under a single path. For example, your application might already use `/info` for another purpose. You can use the `management.contextPath` property to set a prefix for your management endpoint:

```
management.context-path=/manage
```

The `application.properties` example above will change the endpoint from `/{id}` to `/manage/{id}` (e.g. `/manage/info`).

41.3 Customizing the management server port

Exposing management endpoints using the default HTTP port is a sensible choice for cloud based deployments. If, however, your application runs inside your own data center you may prefer to expose endpoints using a different HTTP port.

The `management.port` property can be used to change the HTTP port.

```
management.port=8081
```

Since your management port is often protected by a firewall, and not exposed to the public you might not need security on the management endpoints, even if your main application is secure. In that case you will have Spring Security on the classpath, and you can disable management security like this:

```
management.security.enabled=false
```

(If you don't have Spring Security on the classpath then there is no need to explicitly disable the management security in this way, and it might even break the application.)

41.4 Customizing the management server address

You can customize the address that the management endpoints are available on by setting the `management.address` property. This can be useful if you want to listen only on an internal or ops-facing network, or to only listen for connections from `localhost`.

> **Note**
>
> You can only listen on a different address if the port is different to the main server port.

Here is an example `application.properties` that will not allow remote management connections:

```
management.port=8081
management.address=127.0.0.1
```

41.5 Disabling HTTP endpoints

If you don't want to expose endpoints over HTTP you can set the management port to `-1`:

```
management.port=-1
```

41.6 HTTP Health endpoint access restrictions

The information exposed by the health endpoint varies depending on whether or not it's accessed anonymously. By default, when accessed anonymously, any details about the server's health are hidden and the endpoint will simply indicate whether or not the server is up or down. Furthermore, when accessed anonymously, the response is cached for a configurable period to prevent the endpoint being used in a denial of service attack. The `endpoints.health.time-to-live` property is used to configure the caching period in milliseconds. It defaults to 1000, i.e. one second.

The above-described restrictions can be disabled, thereby allowing anonymous users full access to the health endpoint. To do so, set `endpoints.health.sensitive` to `false`.

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. By default Spring Boot will expose management endpoints as JMX MBeans under the `org.springframework.boot` domain.

42.1 Customizing MBean names

The name of the MBean is usually generated from the `id` of the endpoint. For example the `health` endpoint is exposed as `org.springframework.boot/Endpoint/HealthEndpoint`.

If your application contains more than one Spring `ApplicationContext` you may find that names clash. To solve this problem you can set the `endpoints.jmx.uniqueNames` property to `true` so that MBean names are always unique.

You can also customize the JMX domain under which endpoints are exposed. Here is an example `application.properties`:

```
endpoints.jmx.domain=myapp
endpoints.jmx.uniqueNames=true
```

42.2 Disabling JMX endpoints

If you don't want to expose endpoints over JMX you can set the `spring.jmx.enabled` property to `false`:

```
spring.jmx.enabled=false
```

42.3 Using Jolokia for JMX over HTTP

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

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

Jolokia can then be accessed using `/jolokia` on your management HTTP server.

Customizing Jolokia

Jolokia has a number of settings that you would traditionally configure using servlet parameters. With Spring Boot you can use your `application.properties`, simply prefix the parameter with `jolokia.config.`:

```
jolokia.config.debug=true
```

Disabling Jolokia

If you are using Jolokia but you don't want Spring Boot to configure it, simply set the `endpoints.jolokia.enabled` property to `false`:

```
endpoints.jolokia.enabled=false
```

Spring Boot supports an integrated Java shell called 'CRaSH'. You can use CRaSH to `ssh` or `telnet` into your running application. To enable remote shell support add a dependency to `spring-boot-starter-remote-shell`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-remote-shell</artifactId>
 </dependency>
```

> **Tip**
>
> If you want to also enable telnet access your will additionally need a dependency on `org.crsh:crsh.shell.telnet`.

43.1 Connecting to the remote shell

By default the remote shell will listen for connections on port `2000`. The default user is `user` and the default password will be randomly generated and displayed in the log output. If your application is using Spring Security, the shell will use the same configuration by default. If not, a simple authentication will be applied and you should see a message like this:

```
Using default password for shell access: ec03e16c-4cf4-49ee-b745-7c8255c1dd7e
```

Linux and OSX users can use `ssh` to connect to the remote shell, Windows users can download and install PuTTY.

```
$ ssh -p 2000 user@localhost

user@localhost's password:
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.2.0.RELEASE) on myhost
```

Type `help` for a list of commands. Spring boot provides `metrics`, `beans`, `autoconfig` and `endpoint` commands.

Remote shell credentials

You can use the `shell.auth.simple.user.name` and `shell.auth.simple.user.password` properties to configure custom connection credentials. It is also possible to use a 'Spring Security' `AuthenticationManager` to handle login duties. See the `CrshAutoConfiguration` and `ShellProperties` Javadoc for full details.

43.2 Extending the remote shell

The remote shell can be extended in a number of interesting ways.

Remote shell commands

You can write additional shell commands using Groovy or Java (see the CRaSH documentation for details). By default Spring Boot will search for commands in the following locations:

- `classpath*:/commands/**`
- `classpath*:/crash/commands/**`

> **Tip**
>
> You can change the search path by settings a `shell.commandPathPatterns` property.

Here is a simple 'hello world' command that could be loaded from `src/main/resources/commands/hello.groovy`

```
package commands

import org.crsh.cli.Usage
import org.crsh.cli.Command

class hello {

    @Usage("Say Hello")
    @Command
    def main(InvocationContext context) {
        return "Hello"
    }

}
```

Spring Boot adds some additional attributes to `InvocationContext` that you can access from your command:

Attribute Name	Description
`spring.boot.version`	The version of Spring Boot
`spring.version`	The version of the core Spring Framework
`spring.beanfactory`	Access to the Spring `BeanFactory`
`spring.environment`	Access to the Spring `Environment`

Remote shell plugins

In addition to new commands, it is also possible to extend other CRaSH shell features. All Spring Beans that extend `org.crsh.plugin.CRaSHPlugin` will be automatically registered with the shell.

For more information please refer to the CRaSH reference documentation.

Spring Boot Actuator includes a metrics service with 'gauge' and 'counter' support. A 'gauge' records a single value; and a 'counter' records a delta (an increment or decrement). Spring Boot Actuator also provides a `PublicMetrics` interface that you can implement to expose metrics that you cannot record via one of those two mechanisms. Look at `SystemPublicMetrics` for an example.

Metrics for all HTTP requests are automatically recorded, so if you hit the `metrics` endpoint you should see a response similar to this:

```
{
    "counter.status.200.root": 20,
    "counter.status.200.metrics": 3,
    "counter.status.200.star-star": 5,
    "counter.status.401.root": 4,
    "gauge.response.star-star": 6,
    "gauge.response.root": 2,
    "gauge.response.metrics": 3,
    "classes": 5808,
    "classes.loaded": 5808,
    "classes.unloaded": 0,
    "heap": 3728384,
    "heap.committed": 986624,
    "heap.init": 262144,
    "heap.used": 52765,
    "mem": 986624,
    "mem.free": 933858,
    "processors": 8,
    "threads": 15,
    "threads.daemon": 11,
    "threads.peak": 15,
    "uptime": 494836,
    "instance.uptime": 489782,
    "datasource.primary.active": 5,
    "datasource.primary.usage": 0.25
}
```

Here we can see basic `memory`, `heap`, `class loading`, `processor` and `thread pool` information along with some HTTP metrics. In this instance the `root` ('/') and `/metrics` URLs have returned `HTTP 200` responses `20` and `3` times respectively. It also appears that the `root` URL returned `HTTP 401` (unauthorized) `4` times. The double asterix (`star-star`) comes from a request matched by Spring MVC as `/**` (normally a static resource).

The `gauge` shows the last response time for a request. So the last request to `root` took `2ms` to respond and the last to `/metrics` took `3ms`.

> **Note**
>
> In this example we are actually accessing the endpoint over HTTP using the `/metrics` URL, this explains why `metrics` appears in the response.

44.1 System metrics

The following system metrics are exposed by Spring Boot:

- The total system memory in Kb (`mem`)
- The amount of free memory in Kb (`mem.free`)
- The number of processors (`processors`)

- The system uptime in milliseconds (`uptime`)
- The application context uptime in milliseconds (`instance.uptime`)
- The average system load (`systemload.average`)
- Heap information in Kb (`heap`, `heap.committed`, `heap.init`, `heap.used`)
- Thread information (`threads`, `thread.peak`, `thead.daemon`)
- Class load information (`classes`, `classes.loaded`, `classes.unloaded`)
- Garbage collection information (`gc.xxx.count`, `gc.xxx.time`)

44.2 DataSource metrics

The following metrics are exposed for each supported `DataSource` defined in your application:

- The maximum number connections (`datasource.xxx.max`).
- The minimum number of connections (`datasource.xxx.min`).
- The number of active connections (`datasource.xxx.active`)
- The current usage of the connection pool (`datasource.xxx.usage`).

All data source metrics share the `datasource.` prefix. The prefix is further qualified for each data source:

- If the data source is the primary data source (that is either the only available data source or the one flagged `@Primary` amongst the existing ones), the prefix is `datasource.primary`.
- If the data source bean name ends with `dataSource`, the prefix is the name of the bean without `dataSource` (i.e. `datasource.batch` for `batchDataSource`).
- In all other cases, the name of the bean is used.

It is possible to override part or all of those defaults by registering a bean with a customized version of `DataSourcePublicMetrics`. By default, Spring Boot provides metadata for all supported datasources; you can add additional `DataSourcePoolMetadataProvider` beans if your favorite data source isn't supported out of the box. See `DataSourcePoolMetadataProvidersConfiguration` for examples.

44.3 Tomcat session metrics

If you are using Tomcat as your embedded servlet container, session metrics will automatically be exposed. The `httpsessions.active` and `httpsessions.max` keys provide the number of active and maximum sessions.

44.4 Recording your own metrics

To record your own metrics inject a `CounterService` and/or `GaugeService` into your bean. The `CounterService` exposes `increment`, `decrement` and `reset` methods; the `GaugeService` provides a `submit` method.

Here is a simple example that counts the number of times that a method is invoked:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.CounterService;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    private final CounterService counterService;

    @Autowired
    public MyService(CounterService counterService) {
        this.counterService = counterService;
    }

    public void exampleMethod() {
        this.counterService.increment("services.system.myservice.invoked");
    }

}
```

> **Tip**
>
> You can use any string as a metric name but you should follow guidelines of your chosen store/ graphing technology. Some good guidelines for Graphite are available on Matt Aimonetti's Blog.

44.5 Adding your own public metrics

To add additional metrics that are computed every time the metrics endpoint is invoked, simply register additional `PublicMetrics` implementation bean(s). By default, all such beans are gathered by the endpoint. You can easily change that by defining your own `MetricsEndpoint`.

44.6 Metric repositories

Metric service implementations are usually bound to a `MetricRepository`. A `MetricRepository` is responsible for storing and retrieving metric information. Spring Boot provides an `InMemoryMetricRepository` and a `RedisMetricRepository` out of the box (the in-memory repository is the default) but you can also write your own. The `MetricRepository` interface is actually composed of higher level `MetricReader` and `MetricWriter` interfaces. For full details refer to the Javadoc.

There's nothing to stop you hooking a `MetricRepository` with back-end storage directly into your app, but we recommend using the default `InMemoryMetricRepository` (possibly with a custom `Map` instance if you are worried about heap usage) and populating a back-end repository through a scheduled export job. In that way you get some buffering in memory of the metric values and you can reduce the network chatter by exporting less frequently or in batches. Spring Boot provides an `Exporter` interface and a few basic implementations for you to get started with that.

44.7 Dropwizard Metrics

User of the Dropwizard 'Metrics' library will automatically find that Spring Boot metrics are published to `com.codahale.metrics.MetricRegistry`. A default `com.codahale.metrics.MetricRegistry` Spring bean will be created when you declare a dependency to the `io.dropwizard.metrics:metrics-core` library; you can also register you own `@Bean` instance if you need customizations. Metrics from the `MetricRegistry` are also automatically exposed via the `/metrics` endpoint

Users can create Dropwizard metrics by prefixing their metric names with the appropriate type (e.g. `histogram.*`, `meter.*`).

44.8 Message channel integration

If the 'Spring Messaging' jar is on your classpath a `MessageChannel` called `metricsChannel` is automatically created (unless one already exists). All metric update events are additionally published as 'messages' on that channel. Additional analysis or actions can be taken by clients subscribing to that channel.

Spring Boot Actuator has a flexible audit framework that will publish events once Spring Security is in play (‘authentication success’, ‘failure’ and ‘access denied’ exceptions by default). This can be very useful for reporting, and also to implement a lock-out policy based on authentication failures.

You can also choose to use the audit services for your own business events. To do that you can either inject the existing `AuditEventRepository` into your own components and use that directly, or you can simply publish `AuditApplicationEvent` via the Spring `ApplicationEventPublisher` (using `ApplicationEventPublisherAware`).

Tracing is automatically enabled for all HTTP requests. You can view the `trace` endpoint and obtain basic information about the last few requests:

```
[{
    "timestamp": 1394343677415,
    "info": {
        "method": "GET",
        "path": "/trace",
        "headers": {
            "request": {
                "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
                "Connection": "keep-alive",
                "Accept-Encoding": "gzip, deflate",
                "User-Agent": "Mozilla/5.0 Gecko/Firefox",
                "Accept-Language": "en-US,en;q=0.5",
                "Cookie": "_ga=GA1.1.827067509.1390890128; ..."
                "Authorization": "Basic ...",
                "Host": "localhost:8080"
            },
            "response": {
                "Strict-Transport-Security": "max-age=31536000 ; includeSubDomains",
                "X-Application-Context": "application:8080",
                "Content-Type": "application/json;charset=UTF-8",
                "status": "200"
            }
        }
    }
},{
    "timestamp": 1394343684465,
    ...
}]
```

46.1 Custom tracing

If you need to trace additional events you can inject a `TraceRepository` into your Spring Beans. The `add` method accepts a single `Map` structure that will be converted to JSON and logged.

By default an `InMemoryTraceRepository` will be used that stores the last 100 events. You can define your own instance of the `InMemoryTraceRepository` bean if you need to expand the capacity. You can also create your own alternative `TraceRepository` implementation if needed.

In Spring Boot Actuator you can find a couple of classes to create files that are useful for process monitoring:

- `ApplicationPidFileWriter` creates a file containing the application PID (by default in the application directory with the file name `application.pid`).
- `EmbeddedServerPortFileWriter` creates a file (or files) containing the ports of the embedded server (by default in the application directory with the file name `application.port`).

These writers are not activated by default, but you can enable them in one of the ways described below.

47.1 Extend configuration

In `META-INF/spring.factories` file you have to activate the listener(s):

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.actuate.system.ApplicationPidFileWriter,
org.springframework.boot.actuate.system.EmbeddedServerPortFileWriter
```

47.2 Programmatically

You can also activate a listener by invoking the `SpringApplication.addListeners(…)` method and passing the appropriate `Writer` object. This method also allows you to customize the file name and path via the `Writer` constructor.

If you want to explore some of the concepts discussed in this chapter, you can take a look at the actuator sample applications. You also might want to read about graphing tools such as Graphite.

Otherwise, you can continue on, to read about 'cloud deployment options' or jump ahead for some in-depth information about Spring Boot's *build tool plugins*.

Part VI. Deploying to the cloud

Spring Boot's executable jars are ready-made for most popular cloud PaaS (platform-as-a-service) providers. These providers tend to require that you "bring your own container"; they manage application processes (not Java applications specifically), so they need some intermediary layer that adapts *your* application to the *cloud's* notion of a running process.

Two popular cloud providers, Heroku and Cloud Foundry, employ a "buildpack" approach. The buildpack wraps your deployed code in whatever is needed to *start* your application: it might be a JDK and a call to `java`, it might be an embedded webserver, or it might be a full-fledged application server. A buildpack is pluggable, but ideally you should be able to get by with as few customizations to it as possible. This reduces the footprint of functionality that is not under your control. It minimizes divergence between deployment and production environments.

Ideally, your application, like a Spring Boot executable jar, has everything that it needs to run packaged within it.

In this section we'll look at what it takes to get the simple application that we developed in the "Getting Started" section up and running in the Cloud.

Cloud Foundry provides default buildpacks that come into play if no other buildpack is specified. The Cloud Foundry Java buildpack has excellent support for Spring applications, including Spring Boot. You can deploy stand-alone executable jar applications, as well as traditional `.war` packaged applications.

Once you've built your application (using, for example, `mvn clean package`) and installed the `cf` command line tool, simply deploy your application using the `cf push` command as follows, substituting the path to your compiled `.jar`. Be sure to have logged in with your `cf` command line client before pushing an application.

```
$ cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

See the `cf push` documentation for more options. If there is a Cloud Foundry `manifest.yml` file present in the same directory, it will be consulted.

> **Note**
>
> Here we are substituting `acloudyspringtime` for whatever value you give `cf` as the name of your application.

At this point `cf` will start uploading your application:

```
Uploading acloudyspringtime... OK
Preparing to start acloudyspringtime... OK
-----> Downloaded app package (8.9M)
-----> Java Buildpack source: system
-----> Downloading Open JDK 1.7.0_51 from .../x86_64/openjdk-1.7.0_51.tar.gz (1.8s)
       Expanding Open JDK to .java-buildpack/open_jdk (1.2s)
-----> Downloading Spring Auto Reconfiguration from  0.8.7 .../auto-reconfiguration-0.8.7.jar (0.1s)
-----> Uploading droplet (44M)
Checking status of app 'acloudyspringtime'...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 down)
  ...
  0 of 1 instances running (1 starting)
  ...
  1 of 1 instances running (1 running)

App started
```

Congratulations! The application is now live!

It's easy to then verify the status of the deployed application:

```
$ cf apps
Getting applications in ...
OK

name                requested state   instances   memory   disk   urls
...
acloudyspringtime    started           1/1         512M     1G     acloudyspringtime.cfapps.io
...
```

Once Cloud Foundry acknowledges that your application has been deployed, you should be able to hit the application at the URI given, in this case `acloudyspringtime.cfapps.io/`.

49.1 Binding to services

By default, metadata about the running application as well as service connection information is exposed to the application as environment variables (for example: $VCAP_SERVICES). This architecture decision

is due to Cloud Foundry's polyglot (any language and platform can be supported as a buildpack) nature; process-scoped environment variables are language agnostic.

Environment variables don't always make for the easiest API so Spring Boot automatically extracts them and flattens the data into properties that can be accessed through Spring's `Environment` abstraction:

```
@Component
class MyBean implements EnvironmentAware {

    private String instanceId;

    @Override
    public void setEnvironment(Environment environment) {
        this.instanceId = environment.getProperty("vcap.application.instance_id");
    }

    // ...

}
```

All Cloud Foundry properties are prefixed with `vcap`. You can use vcap properties to access application information (such as the public URL of the application) and service information (such as database credentials). See `VcapApplicationListener` Javdoc for complete details.

> **Tip**
>
> The Spring Cloud Connectors project is a better fit for tasks such as configuring a DataSource. Spring Boot includes auto-configuration support and a `spring-boot-starter-cloud-connectors` starter POM.

Heroku is another popular PaaS platform. To customize Heroku builds, you provide a `Procfile`, which provides the incantation required to deploy an application. Heroku assigns a `port` for the Java application to use and then ensures that routing to the external URI works.

You must configure your application to listen on the correct port. Here's the `Procfile` for our starter REST application:

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
```

Spring Boot makes `-D` arguments available as properties accessible from a Spring `Environment` instance. The `server.port` configuration property is fed to the embedded Tomcat, Jetty or Undertow instance which then uses it when it starts up. The `$PORT` environment variable is assigned to us by the Heroku PaaS.

Heroku by default will use Java 1.6. This is fine as long as your Maven or Gradle build is set to use the same version (Maven users can use the `java.version` property). If you want to use JDK 1.7, create a new file adjacent to your `pom.xml` and `Procfile`, called `system.properties`. In this file add the following:

```
java.runtime.version=1.7
```

This should be everything you need. The most common workflow for Heroku deployments is to `git push` the code to production.

```
$ git push heroku master

Initializing repository, done.
Counting objects: 95, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (78/78), done.
Writing objects: 100% (95/95), 8.66 MiB | 606.00 KiB/s, done.
Total 95 (delta 31), reused 0 (delta 0)

-----> Java app detected
-----> Installing OpenJDK 1.7... done
-----> Installing Maven 3.2.3... done
-----> Installing settings.xml... done
-----> executing /app/tmp/cache/.maven/bin/mvn -B
       -Duser.home=/tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229
       -Dmaven.repo.local=/app/tmp/cache/.m2/repository
       -s /app/tmp/cache/.m2/settings.xml -DskipTests=true clean install

       [INFO] Scanning for projects...
       Downloading: http://repo.spring.io/...
       Downloaded: http://repo.spring.io/... (818 B at 1.8 KB/sec)
        ....
       Downloaded: http://s3pository.heroku.com/jvm/... (152 KB at 595.3 KB/sec)
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/target/...
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/pom.xml ...
       [INFO] ------------------------------------------------------------------------
       [INFO] BUILD SUCCESS
       [INFO] ------------------------------------------------------------------------
       [INFO] Total time: 59.358s
       [INFO] Finished at: Fri Mar 07 07:28:25 UTC 2014
       [INFO] Final Memory: 20M/493M
       [INFO] ------------------------------------------------------------------------

-----> Discovering process types
       Procfile declares types -> web

-----> Compressing... done, 70.4MB
-----> Launching... done, v6
       http://agile-sierra-1405.herokuapp.com/ deployed to Heroku
```

```
To git@heroku.com:agile-sierra-1405.git
 * [new branch]      master -> master
```

Your application should now be up and running on Heroku.

Openshift is the RedHat public (and enterprise) PaaS solution. Like Heroku, it works by running scripts triggered by git commits, so you can script the launching of a Spring Boot application in pretty much any way you like as long as the Java runtime is available (which is a standard feature you can ask for at Openshift). To do this you can use the DIY Cartridge and hooks in your repository under `.openshift/action_scripts`:

The basic model is to:

1. Ensure Java and your build tool are installed remotely, e.g. using a `pre_build` hook (Java and Maven are installed by default, Gradle is not)
2. Use a `build` hook to build your jar (using Maven or Gradle), e.g.

```
#!/bin/bash
cd $OPENSHIFT_REPO_DIR
mvn package -s .openshift/settings.xml -DskipTests=true
```

3. Add a `start` hook that calls `java -jar` …

```
#!/bin/bash
cd $OPENSHIFT_REPO_DIR
nohup java -jar target/*.jar --server.port=${OPENSHIFT_DIY_PORT} --server.address=${OPENSHIFT_DIY_IP}
 &
```

4. Use a `stop` hook (since the start is supposed to return cleanly), e.g.

```
#!/bin/bash
source $OPENSHIFT_CARTRIDGE_SDK_BASH
PID=$(ps -ef | grep java.*\.jar | grep -v grep | awk '{ print $2 }')
if [ -z "$PID" ]
then
    client_result "Application is already stopped"
else
    kill $PID
fi
```

5. Embed service bindings from environment variables provided by the platform in your `application.properties`, e.g.

```
spring.datasource.url: jdbc:mysql://${OPENSHIFT_MYSQL_DB_HOST}:${OPENSHIFT_MYSQL_DB_PORT}/
${OPENSHIFT_APP_NAME}
spring.datasource.username: ${OPENSHIFT_MYSQL_DB_USERNAME}
spring.datasource.password: ${OPENSHIFT_MYSQL_DB_PASSWORD}
```

There's a blog on running Gradle in Openshift on their website that will get you started with a gradle build to run the app. A bug in Gradle currently prevents you from using Gradle newer than 1.6.

Google App Engine is tied to the Servlet 2.5 API, so you can't deploy a Spring Application there without some modifications. See the Servlet 2.5 section of this guide.

Check out the Cloud Foundry, Heroku and Openshift web sites for more information about the kinds of features that a PaaS can offer. These are just three of the most popular Java PaaS providers, since Spring Boot is so amenable to cloud-based deployment you're free to consider other providers as well.

The next section goes on to cover the *Spring Boot CLI*; or you can jump ahead to read about *build tool plugins*.

Part VII. Spring Boot CLI

The Spring Boot CLI is a command line tool that can be used if you want to quickly develop with Spring. It allows you to run Groovy scripts, which means that you have a familiar Java-like syntax, without so much boilerplate code. You can also bootstrap a new project or write your own command for it.

The Spring Boot CLI can be installed manually; using GVM (the Groovy Environment Manually) or using Homebrew if you are an OSX user. See *Section 10.2, "Installing the Spring Boot CLI"* in the "Getting started" section for comprehensive installation instructions.

Once you have installed the CLI you can run it by typing `spring`. If you run `spring` without any arguments, a simple help screen is displayed:

```
$ spring
usage: spring [--help] [--version]
       <command> [<args>]

Available commands are:

  run [options] <files> [--] [args]
    Run a spring groovy script

  ... more command help is shown here
```

You can use `help` to get more details about any of the supported commands. For example:

```
$ spring help run
spring run - Run a spring groovy script

usage: spring run [options] <files> [--] [args]

Option                    Description
------                    -----------
--autoconfigure [Boolean]  Add autoconfigure compiler
                             transformations (default: true)
--classpath, -cp          Additional classpath entries
-e, --edit                Open the file with the default system
                             editor
--no-guess-dependencies   Do not attempt to guess dependencies
--no-guess-imports        Do not attempt to guess imports
-q, --quiet               Quiet logging
-v, --verbose             Verbose logging of dependency
                             resolution
--watch                   Watch the specified file for changes
```

The `version` command provides a quick way to check which version of Spring Boot you are using.

```
$ spring version
Spring CLI v1.2.0.RELEASE
```

55.1 Running applications using the CLI

You can compile and run Groovy source code using the `run` command. The Spring Boot CLI is completely self-contained so you don't need any external Groovy installation.

Here is an example "hello world" web application written in Groovy:

hello.groovy.

```
@RestController
class WebApplication {

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

}
```

To compile and run the application type:

```
$ spring run hello.groovy
```

To pass command line arguments to the application, you need to use a `--` to separate them from the "spring" command arguments, e.g.

```
$ spring run hello.groovy -- --server.port=9000
```

To set JVM command line arguments you can use the `JAVA_OPTS` environment variable, e.g.

```
$ JAVA_OPTS=-Xmx1024m spring run hello.groovy
```

Deduced "grab" dependencies

Standard Groovy includes a `@Grab` annotation which allows you to declare dependencies on a third-party libraries. This useful technique allows Groovy to download jars in the same way as Maven or Gradle would, but without requiring you to use a build tool.

Spring Boot extends this technique further, and will attempt to deduce which libraries to "grab" based on your code. For example, since the `WebApplication` code above uses `@RestController` annotations, "Tomcat" and "Spring MVC" will be grabbed.

The following items are used as "grab hints":

Items	Grabs
`JdbcTemplate`, `NamedParameterJdbcTemplate`, `DataSource`	JDBC Application.
`@EnableJms`	JMS Application.
`@EnableCaching`	Caching abstraction.
`@Test`	JUnit.
`@EnableRabbit`	RabbitMQ.
`@EnableReactor`	Project Reactor.
extends `Specification`	Spock test.
`@EnableBatchProcessing`	Spring Batch.
`@MessageEndpoint` `@EnableIntegrationPatterns`	Spring Integration.
`@EnableDeviceResolver`	Spring Mobile.
`@Controller` `@RestController` `@EnableWebMvc`	Spring MVC + Embedded Tomcat.
`@EnableWebSecurity`	Spring Security.
`@EnableTransactionManagement`	Spring Transaction Management.

> **Tip**
>
> See subclasses of `CompilerAutoConfiguration` in the Spring Boot CLI source code to understand exactly how customizations are applied.

Deduced “grab” coordinates

Spring Boot extends Groovy’s standard `@Grab` support by allowing you to specify a dependency without a group or version, for example `@Grab('freemarker')`. This will consult Spring Boot’s default dependency metadata to deduce the artifact’s group and version. Note that the default metadata is tied to the version of the CLI that you’re using – it will only change when you move to a new version of the CLI, putting you in control of when the versions of your dependencies may change. A table showing the dependencies and their versions that are included in the default metadata can be found in the appendix.

Default import statements

To help reduce the size of your Groovy code, several `import` statements are automatically included. Notice how the example above refers to `@Component`, `@RestController` and `@RequestMapping` without needing to use fully-qualified names or `import` statements.

> **Tip**
>
> Many Spring annotations will work without using `import` statements. Try running your application to see what fails before adding imports.

Automatic main method

Unlike the equivalent Java application, you do not need to include a `public static void main(String[] args)` method with your `Groovy` scripts. A `SpringApplication` is automatically created, with your compiled code acting as the `source`.

Custom “grab” metadata

Spring Boot provides a new `@GrabMetadata` annotation that can be used to provide custom dependency metadata that overrides Spring Boot’s defaults. This metadata is specified by using the annotation to provide coordinates of one or more properties files (deployed to a Maven repository with a “type” identifier of `properties`). Each entry in each properties file must be in the form `group:module=version`.

For example, the following declaration:

```
`@GrabMetadata("com.example.custom-versions:1.0.0")`
```

Will pick up `custom-versions-1.0.0.properties` in a Maven repository under `com/example/custom-versions/1.0.0/`.

Multiple properties files can be specified from the annotation, they will be applied in the order that they’re declared. For example:

```
`@GrabMetadata(["com.example.custom-versions:1.0.0",
       "com.example.more-versions:1.0.0"])`
```

indicates that properties in `more-versions` will override properties in `custom-versions`.

You can use `@GrabMetadata` anywhere that you can use `@Grab`, however, to ensure consistent ordering of the metadata, you can only use `@GrabMetadata` at most once in your application. A useful source of dependency metadata (a superset of Spring Boot) is the Spring IO Platform, e.g. `@GrabMetadata('io.spring.platform:platform-versions:1.0.4.RELEASE')`.

55.2 Testing your code

The `test` command allows you to compile and run tests for your application. Typical usage looks like this:

```
$ spring test app.groovy tests.groovy
Total: 1, Success: 1, : Failures: 0
Passed? true
```

In this example, `tests.groovy` contains JUnit `@Test` methods or Spock `Specification` classes. All the common framework annotations and static methods should be available to you without having to `import` them.

Here is the `tests.groovy` file that we used above (with a JUnit test):

```
class ApplicationTests {

    @Test
    void homeSaysHello() {
        assertEquals("Hello World!", new WebApplication().home())
    }

}
```

> **Tip**
>
> If you have more than one test source files, you might prefer to organize them into a `test` directory.

55.3 Applications with multiple source files

You can use “shell globbing” with all commands that accept file input. This allows you to easily use multiple files from a single directory, e.g.

```
$ spring run *.groovy
```

This technique can also be useful if you want to segregate your “test” or “spec” code from the main application code:

```
$ spring test app/*.groovy test/*.groovy
```

55.4 Packaging your application

You can use the `jar` command to package your application into a self-contained executable jar file. For example:

```
$ spring jar my-app.jar *.groovy
```

The resulting jar will contain the classes produced by compiling the application and all of the application’s dependencies so that it can then be run using `java -jar`. The jar file will also contain entries from the application’s classpath. You can add explicit paths to the jar using `--include` and `--exclude` (both are comma-separated, and both accept prefixes to the values “+” and “-” to signify that they should be removed from the defaults). The default includes are

```
public/**, resources/**, static/**, templates/**, META-INF/**, *
```

and the default excludes are

```
.*, repository/**, build/**, target/**, **/*.jar, **/*.groovy
```

See the output of `spring help jar` for more information.

55.5 Initialize a new project

The `init` command allows you to create a new project using start.spring.io without leaving the shell. For example:

```
$ spring init --dependencies=web,data-jpa my-project
Using service at https://start.spring.io
Project extracted to '/Users/developer/example/my-project'
```

This creates a `my-project` directory with a Maven-based project using `spring-boot-starter-web` and `spring-boot-starter-data-jpa`. You can list the capabilities of the service using the `--list` flag

```
$ spring init --list
=======================================
Capabilities of https://start.spring.io
=======================================

Available dependencies:
-----------------------
actuator - Actuator: Production ready features to help you monitor and manage your application
...
web - Web: Support for full-stack web development, including Tomcat and spring-webmvc
websocket - Websocket: Support for WebSocket development
ws - WS: Support for Spring Web Services

Available project types:
------------------------
gradle-build -  Gradle Config [format:build, build:gradle]
gradle-project -  Gradle Project [format:project, build:gradle]
maven-build -  Maven POM [format:build, build:maven]
maven-project -  Maven Project [format:project, build:maven] (default)

...
```

The `init` command supports many options, check the `help` output for more details. For instance, the following command creates a gradle project using Java 8 and `war` packaging:

```
$ spring init --build=gradle --java-version=1.8 --dependencies=websocket --packaging=war sample-app.zip
Using service at https://start.spring.io
Content saved to 'sample-app.zip'
```

55.6 Using the embedded shell

Spring Boot includes command-line completion scripts for BASH and zsh shells. If you don't use either of these shells (perhaps you are a Windows user) then you can use the `shell` command to launch an integrated shell.

```
$ spring shell
Spring Boot (v1.2.0.RELEASE)
Hit TAB to complete. Type \'help' and hit RETURN for help, and \'exit' to quit.
```

From inside the embedded shell you can run other commands directly:

```
$ version
Spring CLI v1.2.0.RELEASE
```

The embedded shell supports ANSI color output as well as `tab` completion. If you need to run a native command you can use the $ prefix. Hitting `ctrl-c` will exit the embedded shell.

55.7 Adding extensions to the CLI

You can add extensions to the CLI using the `install` command. The command takes one or more sets of artifact coordinates in the format `group:artifact:version`. For example:

```
$ spring install com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

In addition to installing the artifacts identified by the coordinates you supply, all of the artifacts' dependencies will also be installed.

To uninstall a dependency use the `uninstall` command. As with the `install` command, it takes one or more sets of artifact coordinates in the format `group:artifact:version`. For example:

```
$ spring uninstall com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

It will uninstall the artifacts identified by the coordinates you supply and their dependencies.

To uninstall all additional dependencies you can use the `--all` option. For example:

```
$ spring uninstall --all
```

Spring Framework 4.0 has native support for a `beans{}` “DSL” (borrowed from Grails), and you can embed bean definitions in your Groovy application scripts using the same format. This is sometimes a good way to include external features like middleware declarations. For example:

```
@Configuration
class Application implements CommandLineRunner {

    @Autowired
    SharedService service

    @Override
    void run(String... args) {
        println service.message
    }

}

import my.company.SharedService

beans {
    service(SharedService) {
        message = "Hello World"
    }
}
```

You can mix class declarations with `beans{}` in the same file as long as they stay at the top level, or you can put the beans DSL in a separate file if you prefer.

There are some sample groovy scripts available from the GitHub repository that you can use to try out the Spring Boot CLI. There is also extensive javadoc throughout the source code.

If you find that you reach the limit of the CLI tool, you will probably want to look at converting your application to full Gradle or Maven built “groovy project”. The next section covers Spring Boot’s *Build tool plugins* that you can use with Gradle or Maven.

Part VIII. Build tool plugins

Spring Boot provides build tool plugins for Maven and Gradle. The plugins offer a variety of features, including the packaging of executable jars. This section provides more details on both plugins, as well as some help should you need to extend an unsupported build system. If you are just getting started, you might want to read “Chapter 13, *Build systems*” from the Part III, “Using Spring Boot” section first.

The Spring Boot Maven Plugin provides Spring Boot support in Maven, allowing you to package executable jar or war archives and run an application “in-place”. To use it you must be using Maven 3.2 (or better).

> **Note**
>
> Refer to the Spring Boot Maven Plugin Site for complete plugin documentation.

58.1 Including the plugin

To use the Spring Boot Maven Plugin simply include the appropriate XML in the `plugins` section of your `pom.xml`

```
<?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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <!-- ... -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>1.2.0.RELEASE</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

This configuration will repackage a jar or war that is built during the `package` phase of the Maven lifecycle. The following example shows both the repackaged jar, as well as the original jar, in the `target` directory:

```
$ mvn package
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

If you don’t include the `<execution/>` configuration as above, you can run the plugin on its own (but only if the package goal is used as well). For example:

```
$ mvn package spring-boot:repackage
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

If you are using a milestone or snapshot release you will also need to add appropriate `pluginRepository` elements:

```
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <url>http://repo.spring.io/snapshot</url>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
```

```
        <url>http://repo.spring.io/milestone</url>
    </pluginRepository>
</pluginRepositories>
```

58.2 Packaging executable jar and war files

Once `spring-boot-maven-plugin` has been included in your `pom.xml` it will automatically attempt to rewrite archives to make them executable using the `spring-boot:repackage` goal. You should configure your project to build a jar or war (as appropriate) using the usual `packaging` element:

```
<?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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <!-- ... -->
    <packaging>jar</packaging>
    <!-- ... -->
</project>
```

Your existing archive will be enhanced by Spring Boot during the `package` phase. The main class that you want to launch can either be specified using a configuration option, or by adding a `Main-Class` attribute to the manifest in the usual way. If you don't specify a main class the plugin will search for a class with a `public static void main(String[] args)` method.

To build and run a project artifact, you can type the following:

```
$ mvn package
$ java -jar target/mymodule-0.0.1-SNAPSHOT.jar
```

To build a war file that is both executable and deployable into an external container you need to mark the embedded container dependencies as "provided", e.g:

```
<?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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <!-- ... -->
    <packaging>war</packaging>
    <!-- ... -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        <!-- ... -->
    </dependencies>
</project>
```

> **Tip**
>
> See the "Section 74.1, "Create a deployable war file"" section for more details on how to create a deployable war file.

Advanced configuration options and examples are available in the plugin info page.

The Spring Boot Gradle Plugin provides Spring Boot support in Gradle, allowing you to package executable jar or war archives, run Spring Boot applications and omit version information from your `build.gradle` file for “blessed” dependencies.

59.1 Including the plugin

To use the Spring Boot Gradle Plugin simply include a `buildscript` dependency and apply the `spring-boot` plugin:

```
buildscript {
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.2.0.RELEASE")
    }
}
apply plugin: 'spring-boot'
```

If you are using a milestone or snapshot release you will also need to add appropriate `repositories` reference:

```
buildscript {
    repositories {
        maven.url "http://repo.spring.io/snapshot"
        maven.url "http://repo.spring.io/milestone"
    }
    // ...
}
```

59.2 Declaring dependencies without versions

The `spring-boot` plugin will register a custom Gradle `ResolutionStrategy` with your build that allows you to omit version numbers when declaring dependencies to “blessed” artifacts. To make use of this functionality, simply declare dependencies in the usual way, but leave the version number empty:

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    compile("org.thymeleaf:thymeleaf-spring4")
    compile("nz.net.ultraq.thymeleaf:thymeleaf-layout-dialect")
}
```

> **Note**
>
> The version of the `spring-boot` gradle plugin that you declare determines the actual versions of the “blessed” dependencies (this ensures that builds are always repeatable). You should always set the version of the `spring-boot` gradle plugin to the actual Spring Boot version that you wish to use. Details of the versions that are provided can be found in the appendix.

The `spring-boot` plugin will only supply a version where one is not specified. To use a version of an artifact that differs from the one that the plugin would provide, simply specify the version when you declare the dependency as you usually would. For example:

```
dependencies {
    compile("org.thymeleaf:thymeleaf-spring4:2.1.1.RELEASE")
}
```

Custom version management

If is possible to customize the versions used by the `ResolutionStrategy` if you need to deviate from Spring Boot's "blessed" dependencies. Alternative version metadata is consulted using the `versionManagement` configuration. For example:

```
dependencies {
    versionManagement("com.mycorp:mycorp-versions:1.0.0.RELEASE@properties")
    compile("org.springframework.data:spring-data-hadoop")
}
```

Version information needs to be published to a repository as a `.properties` file. For the above example `mycorp-versions.properties` file might contain the following:

```
org.springframework.data\:spring-data-hadoop=2.0.0.RELEASE
```

The properties file takes precedence over Spring Boot's defaults, and can be used to override version numbers if necessary.

59.3 Default exclude rules

Gradle handles "exclude rules" in a slightly different way to Maven which can cause unexpected results when using the starter POMs. Specifically, exclusions declared on a dependency will not be applied when the dependency can be reached through a different path. For example, if a starter POM declares the following:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.0.5.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.0.5.RELEASE</version>
    </dependency>
</dependencies>
```

The `commons-logging` jar will **not** be excluded by Gradle because it is pulled in transitively via `spring-context` (`spring-context` → `spring-core` → `commons-logging`) which does not have an `exclusion` element.

To ensure that correct exclusions are actually applied, the Spring Boot Gradle plugin will automatically add exclusion rules. All exclusions defined in the `spring-boot-dependencies` POM and implicit rules for the "starter" POMs will be added.

If you don't want exclusion rules automatically applied you can use the following configuration:

```
springBoot {
    applyExcludeRules=false
}
```

59.4 Packaging executable jar and war files

Once the `spring-boot` plugin has been applied to your project it will automatically attempt to rewrite archives to make them executable using the `bootRepackage` task. You should configure your project to build a jar or war (as appropriate) in the usual way.

The main class that you want to launch can either be specified using a configuration option, or by adding a `Main-Class` attribute to the manifest. If you don't specify a main class the plugin will search for a class with a `public static void main(String[] args)` method.

To build and run a project artifact, you can type the following:

```
$ gradle build
$ java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar
```

To build a war file that is both executable and deployable into an external container, you need to mark the embedded container dependencies as belonging to a configuration named "providedRuntime", e.g:

```
...
apply plugin: 'war'

war {
    baseName = 'myapp'
    version =  '0.5.0'
}

repositories {
    jcenter()
    maven { url "http://repo.spring.io/libs-snapshot" }
}

configurations {
    providedRuntime
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    providedRuntime("org.springframework.boot:spring-boot-starter-tomcat")
    ...
}
```

> **Tip**
>
> See the "Section 74.1, "Create a deployable war file"" section for more details on how to create a deployable war file.

59.5 Running a project in-place

To run a project in place without building a jar first you can use the "bootRun" task:

```
$ gradle bootRun
```

Running this way makes your static classpath resources (i.e. in `src/main/resources` by default) reloadable in the live application, which can be helpful at development time.

> **Note**
>
> Making static classpath resources reloadable means that `bootRun` does not use the output of the `processResources` task. When invoked using `bootRun` your application will use the resources in their unprocessed form.

59.6 Spring Boot plugin configuration

The gradle plugin automatically extends your build script DSL with a `springBoot` element for global configuration of the Boot plugin. Set the appropriate properties as you would with any other Gradle extension (see below for a list of configuration options):

```
springBoot {
    backupSource = false
}
```

59.7 Repackage configuration

The plugin adds a `bootRepackage` task which you can also configure directly, e.g.:

```
bootRepackage {
    mainClass = 'demo.Application'
}
```

The following configuration options are available:

Name	Description
`enabled`	Boolean flag to switch the repackager off (sometimes useful if you want the other Boot features but not this one)
`mainClass`	The main class that should be run. If not specified the `mainClassName` project property will be used or, if the no `mainClassName` id defined the archive will be searched for a suitable class. "Suitable" means a unique class with a well-formed `main()` method (if more than one is found the build will fail). You should also be able to specify the main class name via the "run" task (`main` property) and/or the "startScripts" (`mainClassName` property) as an alternative to using the "springBoot" configuration.
`classifier`	A file name segment (before the extension) to add to the archive, so that the original is preserved in its original location. Defaults to null in which case the archive is repackaged in place. The default is convenient for many purposes, but if you want to use the original jar as a dependency in another project, it's best to use an extension to define the executable archive.
`withJarTask`	The name or value of the `Jar` task (defaults to all tasks of type `Jar`) which is used to locate the archive to repackage.
`customConfiguration`	The name of the custom configuration whuch is used to populate the nested lib directory (without specifying this you get all compile and runtime dependencies).

59.8 Repackage with custom Gradle configuration

Sometimes it may be more appropriate to not package default dependencies resolved from `compile`, `runtime` and `provided` scopes. If the created executable jar file is intended to be run as it is, you need to have all dependencies nested inside it; however, if the plan is to explode a jar file and run the main class manually, you may already have some of the libraries available via `CLASSPATH`. This is a situation where you can repackage your jar with a different set of dependencies.

Using a custom configuration will automatically disable dependency resolving from `compile`, `runtime` and `provided` scopes. Custom configuration can be either defined globally (inside the `springBoot` section) or per task.

```
task clientJar(type: Jar) {
    appendix = 'client'
    from sourceSets.main.output
    exclude('**/*Something*')
}

task clientBoot(type: BootRepackage, dependsOn: clientJar) {
    withJarTask = clientJar
    customConfiguration = "mycustomconfiguration"
}
```

In above example, we created a new `clientJar` Jar task to package a customized file set from your compiled sources. Then we created a new `clientBoot` BootRepackage task and instructed it to work with only `clientJar` task and `mycustomconfiguration`.

```
configurations {
    mycustomconfiguration.exclude group: 'log4j'
}

dependencies {
    mycustomconfiguration configurations.runtime
}
```

The configuration that we are referring to in `BootRepackage` is a normal Gradle configuration. In the above example we created a new configuration named `mycustomconfiguration` instructing it to derive from a `runtime` and exclude the `log4j` group. If the `clientBoot` task is executed, the repackaged boot jar will have all dependencies from `runtime` but no `log4j` jars.

Configuration options

The following configuration options are available:

Name	Description
`mainClass`	The main class that should be run by the executable archive.
`providedConfiguration`	The name of the provided configuration (defaults to `providedRuntime`).
`backupSource`	If the original source archive should be backed-up before being repackaged (defaults to `true`).
`customConfiguration`	The name of the custom configuration.
`layout`	The type of archive, corresponding to how the dependencies are laid out inside (defaults to a guess based on the archive type).

Name	Description
`requiresUnpack`	A list of dependencies (in the form “groupId:artifactId” that must be unpacked from fat jars in order to run. Items are still packaged into the fat jar, but they will be automatically unpacked when it runs.

59.9 Understanding how the Gradle plugin works

When `spring-boot` is applied to your Gradle project a default task named `bootRepackage` is created automatically. The `bootRepackage` task depends on Gradle `assemble` task, and when executed, it tries to find all jar artifacts whose qualifier is empty (i.e. tests and sources jars are automatically skipped).

Due to the fact that `bootRepackage` finds 'all' created jar artifacts, the order of Gradle task execution is important. Most projects only create a single jar file, so usually this is not an issue; however, if you are planning to create a more complex project setup, with custom `Jar` and `BootRepackage` tasks, there are few tweaks to consider.

If you are 'just' creating custom jar files from your project you can simply disable default `jar` and `bootRepackage` tasks:

```
jar.enabled = false
bootRepackage.enabled = false
```

Another option is to instruct the default `bootRepackage` task to only work with a default `jar` task.

```
bootRepackage.withJarTask = jar
```

If you have a default project setup where the main jar file is created and repackaged, 'and' you still want to create additional custom jars, you can combine your custom repackage tasks together and use `dependsOn` so that the `bootJars` task will run after the default `bootRepackage` task is executed:

```
task bootJars
bootJars.dependsOn = [clientBoot1,clientBoot2,clientBoot3]
build.dependsOn(bootJars)
```

All the above tweaks are usually used to avoid situations where an already created boot jar is repackaged again. Repackaging an existing boot jar will not break anything, but you may find that it includes unnecessary dependencies.

59.10 Publishing artifacts to a Maven repository using Gradle

If you are declaring dependencies without versions and you want to publish artifacts to a Maven repository you will need to configure the Maven publication with details of Spring Boot’s dependency management. This can be achieved by configuring it to publish poms that inherit from `spring-boot-starter-parent` or that import dependency management from `spring-boot-dependencies`. The exact details of this configuration depend on how you’re using Gradle and how you’re trying to publish the artifacts.

Configuring Gradle to produce a pom that inherits dependency management

The following is an example of configuring Gradle to generate a pom that inherits from `spring-boot-starter-parent`. Please refer to the Gradle User Guide for further information.

```
uploadArchives {
    repositories {
        mavenDeployer {
            pom {
                project {
                    parent {
                        groupId "org.springframework.boot"
                        artifactId "spring-boot-starter-parent"
                        version "1.2.0.RELEASE"
                    }
                }
            }
        }
    }
}
```

Configuring Gradle to produce a pom that imports dependency management

The following is an example of configuring Gradle to generate a pom that imports the dependency management provided by `spring-boot-dependencies`. Please refer to the Gradle User Guide for further information.

```
uploadArchives {
    repositories {
        mavenDeployer {
            pom {
                project {
                    dependencyManagement {
                        dependencies {
                            dependency {
                                groupId "org.springframework.boot"
                                artifactId "spring-boot-dependencies"
                                version "1.2.0.RELEASE"
                                type "pom"
                                scope "import"
                            }
                        }
                    }
                }
            }
        }
    }
}
```

If you want to use a build tool other than Maven or Gradle, you will likely need to develop your own plugin. Executable jars need to follow a specific format and certain entries need to be written in an uncompressed form (see the *executable jar format* section in the appendix for details).

The Spring Boot Maven and Gradle plugins both make use of `spring-boot-loader-tools` to actually generate jars. You are also free to use this library directly yourself if you need to.

60.1 Repackaging archives

To repackage an existing archive so that it becomes a self-contained executable archive use `org.springframework.boot.loader.tools.Repackager`. The `Repackager` class takes a single constructor argument that refers to an existing jar or war archive. Use one of the two available `repackage()` methods to either replace the original file or write to a new destination. Various settings can also be configured on the repackager before it is run.

60.2 Nested libraries

When repackaging an archive you can include references to dependency files using the `org.springframework.boot.loader.tools.Libraries` interface. We don't provide any concrete implementations of `Libraries` here as they are usually build system specific.

If your archive already includes libraries you can use `Libraries.NONE`.

60.3 Finding a main class

If you don't use `Repackager.setMainClass()` to specify a main class, the repackager will use ASM to read class files and attempt to find a suitable class with a `public static void main(String[] args)` method. An exception is thrown if more than one candidate is found.

60.4 Example repackage implementation

Here is a typical example repackage:

```
Repackager repackager = new Repackager(sourceJarFile);
repackager.setBackupSource(false);
repackager.repackage(new Libraries() {
            @Override
            public void doWithLibraries(LibraryCallback callback) throws IOException {
                // Build system specific implementation, callback for each dependency
                // callback.library(new Library(nestedFile, LibraryScope.COMPILE));
            }
        });
```

If you're interested in how the build tool plugins work you can look at the `spring-boot-tools` module on GitHub. More technical details of the executable jar format are covered in the appendix.

If you have specific build-related questions you can check out the "how-to" guides.

Part IX. ‘How-to’ guides

This section provides answers to some common ‘how do I do that…’ type of questions that often arise when using Spring Boot. This is by no means an exhaustive list, but it does cover quite a lot.

If you are having a specific problem that we don’t cover here, you might want to check out stackoverflow.com to see if someone has already provided an answer; this is also a great place to ask new questions (please use the `spring-boot` tag).

We’re also more than happy to extend this section; If you want to add a ‘how-to’ you can send us a pull request.

62.1 Troubleshoot auto-configuration

The Spring Boot auto-configuration tries its best to 'do the right thing', but sometimes things fail and it can be hard to tell why.

There is a really useful `AutoConfigurationReport` available in any Spring Boot `ApplicationContext`. You will see it if you enable `DEBUG` logging output. If you use the `spring-boot-actuator` there is also an `autoconfig` endpoint that renders the report in JSON. Use that to debug the application and see what features have been added (and which not) by Spring Boot at runtime.

Many more questions can be answered by looking at the source code and the javadoc. Some rules of thumb:

- Look for classes called `*AutoConfiguration` and read their sources, in particular the `@Conditional*` annotations to find out what features they enable and when. Add `--debug` to the command line or a System property `-Ddebug` to get a log on the console of all the autoconfiguration decisions that were made in your app. In a running Actuator app look at the `autoconfig` endpoint ('/autoconfig' or the JMX equivalent) for the same information.
- Look for classes that are `@ConfigurationProperties` (e.g. `ServerProperties`) and read from there the available external configuration options. The `@ConfigurationProperties` has a `name` attribute which acts as a prefix to external properties, thus `ServerProperties` has `prefix="server"` and its configuration properties are `server.port`, `server.address` etc. In a running Actuator app look at the `configprops` endpoint.
- Look for use of `RelaxedEnvironment` to pull configuration values explicitly out of the `Environment`. It often is used with a prefix.
- Look for `@Value` annotations that bind directly to the `Environment`. This is less flexible than the `RelaxedEnvironment` approach, but does allow some relaxed binding, specifically for OS environment variables (so `CAPITALS_AND_UNDERSCORES` are synonyms for `period.separated`).
- Look for `@ConditionalOnExpression` annotations that switch features on and off in response to SpEL expressions, normally evaluated with place-holders resolved from the `Environment`.

62.2 Customize the Environment or ApplicationContext before it starts

A `SpringApplication` has `ApplicationListeners` and `ApplicationContextInitializers` that are used to apply customizations to the context or environment. Spring Boot loads a number of such customizations for use internally from `META-INF/spring.factories`. There is more than one way to register additional ones:

- Programmatically per application by calling the `addListeners` and `addInitializers` methods on `SpringApplication` before you run it.
- Declaratively per application by setting `context.initializer.classes` or `context.listener.classes`.
- Declaratively for all applications by adding a `META-INF/spring.factories` and packaging a jar file that the applications all use as a library.

The `SpringApplication` sends some special `ApplicationEvents` to the listeners (even some before the context is created), and then registers the listeners for events published by the `ApplicationContext` as well. See *Section 22.4, "Application events and listeners"* in the 'Spring Boot features' section for a complete list.

62.3 Build an ApplicationContext hierarchy (adding a parent or root context)

You can use the `ApplicationBuilder` class to create parent/child `ApplicationContext` hierarchies. See *Section 22.3, "Fluent builder API"* in the 'Spring Boot features' section for more information.

62.4 Create a non-web application

Not all Spring applications have to be web applications (or web services). If you want to execute some code in a `main` method, but also bootstrap a Spring application to set up the infrastructure to use, then it's easy with the `SpringApplication` features of Spring Boot. A `SpringApplication` changes its `ApplicationContext` class depending on whether it thinks it needs a web application or not. The first thing you can do to help it is to just leave the servlet API dependencies off the classpath. If you can't do that (e.g. you are running 2 applications from the same code base) then you can explicitly call `SpringApplication.setWebEnvironment(false)`, or set the `applicationContextClass` property (through the Java API or with external properties). Application code that you want to run as your business logic can be implemented as a `CommandLineRunner` and dropped into the context as a `@Bean` definition.

63.1 Externalize the configuration of SpringApplication

A `SpringApplication` has bean properties (mainly setters) so you can use its Java API as you create the application to modify its behavior. Or you can externalize the configuration using properties in `spring.main.*`. E.g. in `application.properties` you might have.

```
spring.main.web_environment=false
spring.main.show_banner=false
```

and then the Spring Boot banner will not be printed on startup, and the application will not be a web application.

> **Note**
>
> The example above also demonstrates how flexible binding allows the use of underscores (_) as well as dashes (-) in property names.

63.2 Change the location of external properties of an application

By default properties from different sources are added to the Spring `Environment` in a defined order (see *Chapter 23, Externalized Configuration* in the 'Spring Boot features' section for the exact order).

A nice way to augment and modify this is to add `@PropertySource` annotations to your application sources. Classes passed to the `SpringApplication` static convenience methods, and those added using `setSources()` are inspected to see if they have `@PropertySources`, and if they do, those properties are added to the `Environment` early enough to be used in all phases of the `ApplicationContext` lifecycle. Properties added in this way have precedence over any added using the default locations, but have lower priority than system properties, environment variables or the command line.

You can also provide System properties (or environment variables) to change the behavior:

- `spring.config.name` (`SPRING_CONFIG_NAME`), defaults to `application` as the root of the file name.
- `spring.config.location` (`SPRING_CONFIG_LOCATION`) is the file to load (e.g. a classpath resource or a URL). A separate `Environment` property source is set up for this document and it can be overridden by system properties, environment variables or the command line.

No matter what you set in the environment, Spring Boot will always load `application.properties` as described above. If YAML is used then files with the '.yml' extension are also added to the list by default.

See `ConfigFileApplicationListener` for more detail.

63.3 Use 'short' command line arguments

Some people like to use (for example) `--port=9000` instead of `--server.port=9000` to set configuration properties on the command line. You can easily enable this by using placeholders in `application.properties`, e.g.

```
server.port=${port:8080}
```

> **Tip**
>
> If you are inheriting from the `spring-boot-starter-parent` POM, the default filter token of the `maven-resources-plugins` has been changed from `${*}` to `@` (i.e. `@maven.token@` instead of `${maven.token}`) to prevent conflicts with Spring-style placeholders. If you have enabled maven filtering for the `application.properties` directly, you may want to also change the default filter token to use other delimiters.

> **Note**
>
> In this specific case the port binding will work in a PaaS environment like Heroku and Cloud Foundry, since in those two platforms the `PORT` environment variable is set automatically and Spring can bind to capitalized synonyms for `Environment` properties.

63.4 Use YAML for external properties

YAML is a superset of JSON and as such is a very convenient syntax for storing external properties in a hierarchical format. E.g.

```
spring:
    application:
        name: cruncher
    datasource:
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost/test
server:
    port: 9000
```

Create a file called `application.yml` and stick it in the root of your classpath, and also add `snakeyaml` to your dependencies (Maven coordinates `org.yaml:snakeyaml`, already included if you use the `spring-boot-starter`). A YAML file is parsed to a Java `Map<String,Object>` (like a JSON object), and Spring Boot flattens the map so that it is 1-level deep and has period-separated keys, a lot like people are used to with `Properties` files in Java.

The example YAML above corresponds to an `application.properties` file

```
spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

See *Section 23.5, "Using YAML instead of Properties"* in the 'Spring Boot features' section for more information about YAML.

63.5 Set the active Spring profiles

The Spring `Environment` has an API for this, but normally you would set a System profile (`spring.profiles.active`) or an OS environment variable (`SPRING_PROFILES_ACTIVE`). E.g. launch your application with a `-D` argument (remember to put it before the main class or jar archive):

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

In Spring Boot you can also set the active profile in `application.properties`, e.g.

```
spring.profiles.active=production
```

A value set this way is replaced by the System property or environment variable setting, but not by the `SpringApplicationBuilder.profiles()` method. Thus the latter Java API can be used to augment the profiles without changing the defaults.

See *Chapter 24, Profiles* in the 'Spring Boot features' section for more information.

63.6 Change configuration depending on the environment

A YAML file is actually a sequence of documents separated by `---` lines, and each document is parsed separately to a flattened map.

If a YAML document contains a `spring.profiles` key, then the profiles value (comma-separated list of profiles) is fed into the Spring `Environment.acceptsProfiles()` and if any of those profiles is active that document is included in the final merge (otherwise not).

Example:

```
server:
    port: 9000
---

spring:
    profiles: development
server:
    port: 9001

---

spring:
    profiles: production
server:
    port: 0
```

In this example the default port is 9000, but if the Spring profile 'development' is active then the port is 9001, and if 'production' is active then it is 0.

The YAML documents are merged in the order they are encountered (so later values override earlier ones).

To do the same thing with properties files you can use `application-${profile}.properties` to specify profile-specific values.

63.7 Discover built-in options for external properties

Spring Boot binds external properties from `application.properties` (or `.yml`) (and other places) into an application at runtime. There is not (and technically cannot be) an exhaustive list of all supported properties in a single location because contributions can come from additional jar files on your classpath.

A running application with the Actuator features has a `configprops` endpoint that shows all the bound and bindable properties available through `@ConfigurationProperties`.

The appendix includes an `application.properties` example with a list of the most common properties supported by Spring Boot. The definitive list comes from searching the source code for `@ConfigurationProperties` and `@Value` annotations, as well as the occasional use of `RelaxedEnvironment`.

64.1 Add a Servlet, Filter or ServletContextListener to an application

`Servlet`, `Filter`, `ServletContextListener` and the other listeners supported by the Servlet spec can be added to your application as `@Bean` definitions. Be very careful that they don't cause eager initialization of too many other beans because they have to be installed in the container very early in the application lifecycle (e.g. it's not a good idea to have them depend on your `DataSource` or JPA configuration). You can work around restrictions like that by initializing them lazily when first used instead of on initialization.

In the case of `Filters` and `Servlets` you can also add mappings and init parameters by adding a `FilterRegistrationBean` or `ServletRegistrationBean` instead of or as well as the underlying component.

64.2 Change the HTTP port

In a standalone application the main HTTP port defaults to `8080`, but can be set with `server.port` (e.g. in `application.properties` or as a System property). Thanks to relaxed binding of `Environment` values you can also use `SERVER_PORT` (e.g. as an OS environment variable).

To switch off the HTTP endpoints completely, but still create a `WebApplicationContext`, use `server.port=-1` (this is sometimes useful for testing).

For more details look at *the section called "Customizing embedded servlet containers"* in the 'Spring Boot features' section, or the `ServerProperties` source code.

64.3 Use a random unassigned HTTP port

To scan for a free port (using OS natives to prevent clashes) use `server.port=0`.

64.4 Discover the HTTP port at runtime

You can access the port the server is running on from log output or from the `EmbeddedWebApplicationContext` via its `EmbeddedServletContainer`. The best way to get that and be sure that it has initialized is to add a `@Bean` of type `ApplicationListener<EmbeddedServletContainerInitializedEvent>` and pull the container out of the event when it is published.

A useful practice for use with `@IntegrationTests` is to set `server.port=0` and then inject the actual ('local') port as a `@Value`. For example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SampleDataJpaApplication.class)
@WebAppConfiguration
@IntegrationTest("server.port:0")
public class CityRepositoryIntegrationTests {

    @Autowired
    EmbeddedWebApplicationContext server;

    @Value("${local.server.port}")
    int port;

    // ...

}
```

64.5 Configure SSL

SSL can be configured declaratively by setting the various `server.ssl.*` properties, typically in `application.properties` or `application.yml`. For example:

```
server.port = 8443
server.ssl.key-store = classpath:keystore.jks
server.ssl.key-store-password = secret
server.ssl.key-password = another-secret
```

See `Ssl` for details of all of the supported properties.

> **Note**
>
> Tomcat requires the key store (and trust store if you're using one) to be directly accessible on the filesystem, i.e. it cannot be read from within a jar file.

64.6 Configure Tomcat

Generally you can follow the advice from *Section 63.7, "Discover built-in options for external properties"* about `@ConfigurationProperties` (`ServerProperties` is the main one here), but also look at `EmbeddedServletContainerCustomizer` and various Tomcat-specific `*Customizers` that you can add in one of those. The Tomcat APIs are quite rich so once you have access to the `TomcatEmbeddedServletContainerFactory` you can modify it in a number of ways. Or the nuclear option is to add your own `TomcatEmbeddedServletContainerFactory`.

64.7 Enable Multiple Connectors with Tomcat

Add a `org.apache.catalina.connector.Connector` to the `TomcatEmbeddedServletContainerFactory` which can allow multiple connectors, e.g. HTTP and HTTPS connector:

```
@Bean
public EmbeddedServletContainerFactory servletContainer() {
    TomcatEmbeddedServletContainerFactory tomcat = new TomcatEmbeddedServletContainerFactory();
    tomcat.addAdditionalTomcatConnectors(createSslConnector());
    return tomcat;
}

private Connector createSslConnector() {
    Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
    Http11NioProtocol protocol = (Http11NioProtocol) connector.getProtocolHandler();
    try {
        File keystore = new ClassPathResource("keystore").getFile();
        File truststore = new ClassPathResource("keystore").getFile();
        connector.setScheme("https");
        connector.setSecure(true);
        connector.setPort(8443);
        protocol.setSSLEnabled(true);
        protocol.setKeystoreFile(keystore.getAbsolutePath());
        protocol.setKeystorePass("changeit");
        protocol.setTruststoreFile(truststore.getAbsolutePath());
        protocol.setTruststorePass("changeit");
        protocol.setKeyAlias("apitester");
        return connector;
    }
    catch (IOException ex) {
        throw new IllegalStateException("can't access keystore: [" + "keystore"
                + "] or truststore: [" + "keystore" + "]", ex);
    }
```

```
}
```

64.8 Use Tomcat behind a front-end proxy server

Spring Boot will automatically configure Tomcat's `RemoteIpValve` if you enable it. This allows you to transparently use the standard `x-forwarded-for` and `x-forwarded-proto` headers that most front-end proxy servers add. The valve is switched on by setting one or both of these properties to something non-empty (these are the conventional values used by most proxies, and if you only set one the other will be set automatically):

```
server.tomcat.remote_ip_header=x-forwarded-for
server.tomcat.protocol_header=x-forwarded-proto
```

If your proxy uses different headers you can customize the valve's configuration by adding some entries to `application.properties`, e.g.

```
server.tomcat.remote_ip_header=x-your-remote-ip-header
server.tomcat.protocol_header=x-your-protocol-header
```

The valve is also configured with a default regular expression that matches internal proxies that are to be trusted. By default, IP addresses in 10/8, 192.168/16, 169.254/16 and 127/8 are trusted. You can customize the valve's configuration by adding an entry to `application.properties`, e.g.

```
server.tomcat.internal_proxies=192\\.168\\.\\d{1,3}\\.\\d{1,3}
```

> **Note**
>
> The double backslashes are only required when you're using a properties file for configuration. If you are using YAML, single backslashes are sufficient and a value that's equivalent to the one shown above would be `192\.168\.\d{1,3}\.\d{1,3}`.

Alternatively, you can take complete control of the configuration of the `RemoteIpValve` by configuring and adding it in a `TomcatEmbeddedServletContainerFactory` bean.

64.9 Use Jetty instead of Tomcat

The Spring Boot starters (`spring-boot-starter-web` in particular) use Tomcat as an embedded container by default. You need to exclude those dependencies and include the Jetty one instead. Spring Boot provides Tomcat and Jetty dependencies bundled together as separate starters to help make this process as easy as possible.

Example in Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
```

```
</dependency>
```

Example in Gradle:

```
configurations {
    compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.2.0.RELEASE")
    compile("org.springframework.boot:spring-boot-starter-jetty:1.2.0.RELEASE")
    // ...
}
```

64.10 Configure Jetty

Generally you can follow the advice from *Section 63.7, "Discover built-in options for external properties"* about `@ConfigurationProperties` (`ServerProperties` is the main one here), but also look at `EmbeddedServletContainerCustomizer`. The Jetty APIs are quite rich so once you have access to the `JettyEmbeddedServletContainerFactory` you can modify it in a number of ways. Or the nuclear option is to add your own `JettyEmbeddedServletContainerFactory`.

64.11 Use Undertow instead of Tomcat

Using Undertow instead of Tomcat is very similar to using Jetty instead of Tomcat. You need to exclude the Tomcat dependencies and include the Undertow starter instead.

Example in Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

Example in Gradle:

```
configurations {
    compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.2.0.RELEASE")
    compile("org.springframework.boot:spring-boot-starter-undertow:1.2.0.RELEASE")
    // ...
}
```

64.12 Configure Undertow

Generally you can follow the advice from *Section 63.7, "Discover built-in options for external properties"* about `@ConfigurationProperties` (`ServerProperties` and `ServerProperties.Undertow` are the main ones here), but also look at `EmbeddedServletContainerCustomizer`. Once

you have access to the `UndertowEmbeddedServletContainerFactory` you can use an `UndertowBuilderCustomizer` to modify Undertow's configuration to meet your needs. Or the nuclear option is to add your own `UndertowEmbeddedServletContainerFactory`.

64.13 Use Tomcat 7

Tomcat 7 works with Spring Boot, but the default is to use Tomcat 8. If you cannot use Tomcat 8 (for example, because you are using Java 1.6) you will need to change your classpath to reference Tomcat 7 and Servlet API 3.0.

If you are using the starter poms and parent you can just change the version properties, e.g. for a simple webapp or service:

```
<properties>
    <tomcat.version>7.0.56</tomcat.version>
    <servlet-api.version>3.0.1</servlet-api.version>
</properties>
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    ...
</dependencies>
```

64.14 Use Jetty 8

Jetty 8 works with Spring Boot, but the default is to use Jetty 9. If you cannot use Jetty 9 (for example, because you are using Java 1.6) you will need to change your classpath to reference Jetty 8 and Servlet API 3.0. You will also need to exclude Jetty's WebSocket-related dependencies.

If you are using the starter poms and parent you can just add the Jetty starter with the required WebSocket exclusion and change the version properties, e.g. for a simple webapp or service:

```
<properties>
    <jetty.version>8.1.15.v20140411</jetty.version>
    <jetty-jsp.version>2.2.0.v201112011158</jetty-jsp.version>
    <servlet-api.version>3.0.1</servlet-api.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.eclipse.jetty.websocket</groupId>
                <artifactId>*</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

64.15 Create WebSocket endpoints using @ServerEndpoint

If you want to use `@ServerEndpoint` in a Spring Boot application that used an embedded container, you must declare a single `ServerEndpointExporter` `@Bean`:

```
@Bean
public ServerEndpointExporter serverEndpointExporter() {
    return new ServerEndpointExporter();
}
```

This bean will register any `@ServerEndpoint` annotated beans with the underlying WebSocket container. When deployed to a standalone servlet container this role is performed by a servlet container initializer and the `ServerEndpointExporter` bean is not required.

65.1 Write a JSON REST service

Any Spring `@RestController` in a Spring Boot application should render JSON response by default as long as Jackson2 is on the classpath. For example:

```
@RestController
public class MyController {

    @RequestMapping("/thing")
    public MyThing thing() {
            return new MyThing();
    }

}
```

As long as `MyThing` can be serialized by Jackson2 (e.g. a normal POJO or Groovy object) then `localhost:8080/thing` will serve a JSON representation of it by default. Sometimes in a browser you might see XML responses because browsers tend to send accept headers that prefer XML.

65.2 Write an XML REST service

If you have the Jackson XML extension (`jackson-dataformat-xml`) on the classpath, it will be used to render XML responses and the very same example as we used for JSON would work. To use it, add the following dependency to your project:

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

You may also want to add a dependency on Woodstox. It's faster than the default Stax implementation provided by the JDK and also adds pretty print support and improved namespace handling:

```
<dependency>
    <groupId>org.codehaus.woodstox</groupId>
    <artifactId>woodstox-core-asl</artifactId>
</dependency>
```

If Jackson's XML extension is not available, JAXB (provided by default in the JDK) will be used, with the additional requirement to have `MyThing` annotated as `@XmlRootElement`:

```
@XmlRootElement
public class MyThing {
    private String name;
    // .. getters and setters
}
```

To get the server to render XML instead of JSON you might have to send an `Accept: text/xml` header (or use a browser).

65.3 Customize the Jackson ObjectMapper

Spring MVC (client and server side) uses `HttpMessageConverters` to negotiate content conversion in an HTTP exchange. If Jackson is on the classpath you already get the default converter(s) provided by `Jackson2ObjectMapperBuilder`.

The `ObjectMapper` (or `XmlMapper` for Jackson XML converter) instance created by default have the following customized properties:

- `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled

Spring Boot has also some features to make it easier to customize this behavior.

You can configure the `ObjectMapper` and `XmlMapper` instances using the environment. Jackson provides an extensive suite of simple on/off features that can be used to configure various aspects of its processing. These features are described in five enums in Jackson which map onto properties in the environment:

Jackson enum	Environment property
`com.fasterxml.jackson.databind.DeserializationFeature`	`spring.jackson.deserialization.<feature_name>=true|false`
`com.fasterxml.jackson.core.JsonGenerator.Feature`	`spring.jackson.generator.<feature_name>=true|false`
`com.fasterxml.jackson.databind.MapperFeature`	`spring.jackson.mapper.<feature_name>=true|false`
`com.fasterxml.jackson.core.JsonParser.Feature`	`spring.jackson.parser.<feature_name>=true|false`
`com.fasterxml.jackson.databind.SerializationFeature`	`spring.jackson.serialization.<feature_name>=true|false`

For example, to enable pretty print, set `spring.jackson.serialization.indent_output=true`. Note that, thanks to the use of relaxed binding, the case of `indent_output` doesn't have to match the case of the corresponding enum constant which is `INDENT_OUTPUT`.

If you want to replace the default `ObjectMapper` completely, define a `@Bean` of that type and mark it as `@Primary`.

Defining a `@Bean` of type `Jackson2ObjectMapperBuilder` will allow you to customize both default `ObjectMapper` and `XmlMapper` (used in `MappingJackson2HttpMessageConverter` and `MappingJackson2XmlHttpMessageConverter` respectively).

Another way to customize Jackson is to add beans of type `com.fasterxml.jackson.databind.Module` to your context. They will be registered with every bean of type `ObjectMapper`, providing a global mechanism for contributing custom modules when you add new features to your application.

Finally, if you provide any `@Beans` of type `MappingJackson2HttpMessageConverter` then they will replace the default value in the MVC configuration. Also, a convenience bean is provided of type `HttpMessageConverters` (always available if you use the default MVC configuration) which has some useful methods to access the default and user-enhanced message converters.

See also the *Section 65.4, "Customize the @ResponseBody rendering"* section and the `WebMvcAutoConfiguration` source code for more details.

65.4 Customize the @ResponseBody rendering

Spring uses `HttpMessageConverters` to render `@ResponseBody` (or responses from `@RestController`). You can contribute additional converters by simply adding beans of that type in a Spring Boot context. If a bean you add is of a type that would have been included by default anyway (like `MappingJackson2HttpMessageConverter` for JSON conversions) then it will replace the default value. A convenience bean is provided of type `HttpMessageConverters` (always available if you use the default MVC configuration) which has some useful methods to access the default and user-enhanced message converters (useful, for example if you want to manually inject them into a custom `RestTemplate`).

As in normal MVC usage, any `WebMvcConfigurerAdapter` beans that you provide can also contribute converters by overriding the `configureMessageConverters` method, but unlike with normal MVC, you can supply only additional converters that you need (because Spring Boot uses the same mechanism to contribute its defaults). Finally, if you opt-out of the Spring Boot default MVC configuration by providing your own `@EnableWebMvc` configuration, then you can take control completely and do everything manually using `getMessageConverters` from `WebMvcConfigurationSupport`.

See the `WebMvcAutoConfiguration` source code for more details.

65.5 Handling Multipart File Uploads

Spring Boot embraces the Servlet 3 `javax.servlet.http.Part` API to support uploading files. By default Spring Boot configures Spring MVC with a maximum file of 1Mb per file and a maximum of 10Mb of file data in a single request. You may override these values, as well as the location to which intermediate data is stored (e.g., to the `/tmp` directory) and the threshold past which data is flushed to disk by using the properties exposed in the `MultipartProperties` class. If you want to specify that files be unlimited, for example, set the `multipart.maxFileSize` property to `-1`.

The multipart support is helpful when you want to receive multipart encoded file data as a `@RequestParam`-annotated parameter of type `MultipartFile` in a Spring MVC controller handler method.

See the `MultipartAutoConfiguration` source for more details.

65.6 Switch off the Spring MVC DispatcherServlet

Spring Boot wants to serve all content from the root of your application `/` down. If you would rather map your own servlet to that URL you can do it, but of course you may lose some of the other Boot MVC features. To add your own servlet and map it to the root resource just declare a `@Bean` of type `Servlet` and give it the special bean name `dispatcherServlet` (You can also create a bean of a different type with that name if you want to switch it off and not replace it).

65.7 Switch off the Default MVC configuration

The easiest way to take complete control over MVC configuration is to provide your own `@Configuration` with the `@EnableWebMvc` annotation. This will leave all MVC configuration in your hands.

65.8 Customize ViewResolvers

A `ViewResolver` is a core component of Spring MVC, translating view names in `@Controller` to actual `View` implementations. Note that `ViewResolvers` are mainly used in UI applications, rather than REST-style services (a `View` is not used to render a `@ResponseBody`). There are many implementations of `ViewResolver` to choose from, and Spring on its own is not opinionated about which ones you should use. Spring Boot, on the other hand, installs one or two for you depending on what it finds on the classpath and in the application context. The `DispatcherServlet` uses all the resolvers it finds in the application context, trying each one in turn until it gets a result, so if you are adding your own you have to be aware of the order and in which position your resolver is added.

`WebMvcAutoConfiguration` adds the following `ViewResolvers` to your context:

- An `InternalResourceViewResolver` with bean id 'defaultViewResolver'. This one locates physical resources that can be rendered using the `DefaultServlet` (e.g. static resources and JSP pages if you are using those). It applies a prefix and a suffix to the view name and then looks for a physical resource with that path in the servlet context (defaults are both empty, but accessible for external configuration via `spring.view.prefix` and `spring.view.suffix`). It can be overridden by providing a bean of the same type.
- A `BeanNameViewResolver` with id 'beanNameViewResolver'. This is a useful member of the view resolver chain and will pick up any beans with the same name as the `View` being resolved. It shouldn't be necessary to override or replace it.
- A `ContentNegotiatingViewResolver` with id 'viewResolver' is only added if there **are** actually beans of type `View` present. This is a 'master' resolver, delegating to all the others and attempting to find a match to the 'Accept' HTTP header sent by the client. There is a useful blog about `ContentNegotiatingViewResolver` that you might like to study to learn more, and also look at the source code for detail. You can switch off the auto-configured `ContentNegotiatingViewResolver` by defining a bean named 'viewResolver'.
- If you use Thymeleaf you will also have a `ThymeleafViewResolver` with id 'thymeleafViewResolver'. It looks for resources by surrounding the view name with a prefix and suffix (externalized to `spring.thymeleaf.prefix` and `spring.thymeleaf.suffix`, defaults 'classpath:/templates/' and '.html' respectively). It can be overridden by providing a bean of the same name.
- If you use FreeMarker you will also have a `FreeMarkerViewResolver` with id 'freeMarkerViewResolver'. It looks for resources in a loader path (externalized to `spring.freemarker.templateLoaderPath`, default 'classpath:/templates/') by surrounding the view name with a prefix and suffix (externalized to `spring.freemarker.prefix` and `spring.freemarker.suffix`, with empty and '.ftl' defaults respectively). It can be overridden by providing a bean of the same name.
- If you use Groovy templates (actually if groovy-templates is on your classpath) you will also have a `Groovy TemplateViewResolver` with id 'groovyTemplateViewResolver'. It looks for resources in a loader path by surrounding the view name with a prefix and suffix (externalized to `spring.groovy.template.prefix` and `spring.groovy.template.suffix`, defaults 'classpath:/templates/' and '.tpl' respectively). It can be overriden by providing a bean of the same name.
- If you use Velocity you will also have a `VelocityViewResolver` with id 'velocityViewResolver'. It looks for resources in a loader path (externalized to `spring.velocity.resourceLoaderPath`,

default ‘classpath:/templates/’) by surrounding the view name with a prefix and suffix (externalized to `spring.velocity.prefix` and `spring.velocity.suffix`, with empty and ‘.vm’ defaults respectively). It can be overridden by providing a bean of the same name.

Check out `WebMvcAutoConfiguration`, `ThymeleafAutoConfiguration`, `FreeMarkerAutoConfiguration`, `GroovyTemplateAutoConfiguration` and `VelocityAutoConfiguration`

Spring Boot has no mandatory logging dependence, except for the `commons-logging` API, of which there are many implementations to choose from. To use Logback you need to include it, and some bindings for `commons-logging` on the classpath. The simplest way to do that is through the starter poms which all depend on `spring-boot-starter-logging`. For a web application you only need `spring-boot-starter-web` since it depends transitively on the logging starter. For example, using Maven:

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

Spring Boot has a `LoggingSystem` abstraction that attempts to configure logging based on the content of the classpath. If Logback is available it is the first choice.

If the only change you need to make to logging is to set the levels of various loggers then you can do that in `application.properties` using the "logging.level" prefix, e.g.

```
logging.level.org.springframework.web: DEBUG
logging.level.org.hibernate: ERROR
```

You can also set the location of a file to log to (in addition to the console) using "logging.file".

To configure the more fine-grained settings of a logging system you need to use the native configuration format supported by the `LoggingSystem` in question. By default Spring Boot picks up the native configuration from its default location for the system (e.g. `classpath:logback.xml` for Logback), but you can set the location of the config file using the "logging.config" property.

66.1 Configure Logback for logging

If you put a `logback.xml` in the root of your classpath it will be picked up from there. Spring Boot provides a default base configuration that you can include if you just want to set levels, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>
    <logger name="org.springframework.web" level="DEBUG"/>
</configuration>
```

If you look at the default `logback.xml` in the spring-boot jar you will see that it uses some useful System properties which the `LoggingSystem` takes care of creating for you. These are:

- `${PID}` the current process ID.
- `${LOG_FILE}` if `logging.file` was set in Boot's external configuration.
- `${LOG_PATH}` if `logging.path` was set (representing a directory for log files to live in).

Spring Boot also provides some nice ANSI colour terminal output on a console (but not in a log file) using a custom Logback converter. See the default `base.xml` configuration for details.

If Groovy is on the classpath you should be able to configure Logback with `logback.groovy` as well (it will be given preference if present).

66.2 Configure Log4j for logging

Spring Boot also supports either Log4j or Log4j 2 for logging configuration, but only if one of them is on the classpath. If you are using the starter poms for assembling dependencies that means you have to exclude Logback and then include your chosen version of Log4j instead. If you aren't using the starter poms then you need to provide `commons-logging` (at least) in addition to your chosen version of Log4j.

The simplest path is probably through the starter poms, even though it requires some jiggling with excludes, .e.g. in Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j</artifactId>
</dependency>
```

To use Log4j 2, simply depend on `spring-boot-starter-log4j2` rather than `spring-boot-starter-log4j`.

> **Note**
>
> The use of one of the Log4j starters gathers together the dependencies for common logging requirements (e.g. including having Tomcat use `java.util.logging` but configuring the output using Log4j or Log4j 2). See the Actuator Log4j or Log4j 2 samples for more detail and to see it in action.

67.1 Configure a DataSource

To override the default settings just define a `@Bean` of your own of type `DataSource`. Spring Boot provides a utility builder class `DataSourceBuilder` that can be used to create one of the standard ones (if it is on the classpath), or you can just create your own, and bind it to a set of `Environment` properties e.g.

```
@Bean
@ConfigurationProperties(prefix="datasource.mine")
public DataSource dataSource() {
    return new FancyDataSource();
}
```

```
datasource.mine.jdbcUrl=jdbc:h2:mem:mydb
datasource.mine.user=sa
datasource.mine.poolSize=30
```

See *Section 28.1, "Configure a DataSource"* in the 'Spring Boot features' section and the `DataSourceAutoConfiguration` class for more details.

67.2 Configure Two DataSources

Creating more than one data source works the same as creating the first one. You might want to mark one of them as `@Primary` if you are using the default auto-configuration for JDBC or JPA (then that one will be picked up by any `@Autowired` injections).

```
@Bean
@Primary
@ConfigurationProperties(prefix="datasource.primary")
public DataSource primaryDataSource() {
    return DataSourceBuilder.create().build();
}

@Bean
@ConfigurationProperties(prefix="datasource.secondary")
public DataSource secondaryDataSource() {
    return DataSourceBuilder.create().build();
}
```

67.3 Use Spring Data repositories

Spring Data can create implementations for you of `@Repository` interfaces of various flavors. Spring Boot will handle all of that for you as long as those `@Repositories` are included in the same package (or a sub-package) of your `@EnableAutoConfiguration` class.

For many applications all you will need is to put the right Spring Data dependencies on your classpath (there is a `spring-boot-starter-data-jpa` for JPA and a `spring-boot-starter-data-mongodb` for Mongodb), create some repository interfaces to handle your `@Entity` objects. Examples are in the JPA sample or the Mongodb sample.

Spring Boot tries to guess the location of your `@Repository` definitions, based on the `@EnableAutoConfiguration` it finds. To get more control, use the `@EnableJpaRepositories` annotation (from Spring Data JPA).

67.4 Separate @Entity definitions from Spring configuration

Spring Boot tries to guess the location of your `@Entity` definitions, based on the `@EnableAutoConfiguration` it finds. To get more control, you can use the `@EntityScan` annotation, e.g.

```
@Configuration
@EnableAutoConfiguration
@EntityScan(basePackageClasses=City.class)
public class Application {

    //...

}
```

67.5 Configure JPA properties

Spring Data JPA already provides some vendor-independent configuration options (e.g. for SQL logging) and Spring Boot exposes those, and a few more for hibernate as external configuration properties. The most common options to set are:

```
spring.jpa.hibernate.ddl-auto: create-drop
spring.jpa.hibernate.naming_strategy: org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.database: H2
spring.jpa.show-sql: true
```

(Because of relaxed data binding hyphens or underscores should work equally well as property keys.) The `ddl-auto` setting is a special case in that it has different defaults depending on whether you are using an embedded database (`create-drop`) or not (`none`). In addition all properties in `spring.jpa.properties.*` are passed through as normal JPA properties (with the prefix stripped) when the local `EntityManagerFactory` is created.

See `HibernateJpaAutoConfiguration` and `JpaBaseConfiguration` for more details.

67.6 Use a custom EntityManagerFactory

To take full control of the configuration of the `EntityManagerFactory`, you need to add a `@Bean` named ‘entityManagerFactory’. Spring Boot auto-configuration switches off its entity manager based on the presence of a bean of that type.

67.7 Use Two EntityManagers

Even if the default `EntityManagerFactory` works fine, you will need to define a new one because otherwise the presence of the second bean of that type will switch off the default. To make it easy to do that you can use the convenient `EntityManagerBuilder` provided by Spring Boot, or if you prefer you can just use the `LocalContainerEntityManagerFactoryBean` directly from Spring ORM.

Example:

```
// add two data sources configured as above

@Bean
public LocalContainerEntityManagerFactoryBean customerEntityManagerFactory(
        EntityManagerFactoryBuilder builder) {
    return builder
            .dataSource(customerDataSource())
            .packages(Customer.class)
```

```
            .persistenceUnit("customers")
            .build();
}

@Bean
public LocalContainerEntityManagerFactoryBean orderEntityManagerFactory(
        EntityManagerFactoryBuilder builder) {
    return builder
            .dataSource(orderDataSource())
            .packages(Order.class)
            .persistenceUnit("orders")
            .build();
}
```

The configuration above almost works on its own. To complete the picture you need to configure `TransactionManagers` for the two `EntityManagers` as well. One of them could be picked up by the default `JpaTransactionManager` in Spring Boot if you mark it as `@Primary`. The other would have to be explicitly injected into a new instance. Or you might be able to use a JTA transaction manager spanning both.

67.8 Use a traditional persistence.xml

Spring doesn't require the use of XML to configure the JPA provider, and Spring Boot assumes you want to take advantage of that feature. If you prefer to use `persistence.xml` then you need to define your own `@Bean` of type `LocalEntityManagerFactoryBean` (with id 'entityManagerFactory', and set the persistence unit name there.

See `JpaBaseConfiguration` for the default settings.

67.9 Use Spring Data JPA and Mongo repositories

Spring Data JPA and Spring Data Mongo can both create `Repository` implementations for you automatically. If they are both present on the classpath, you might have to do some extra configuration to tell Spring Boot which one (or both) you want to create repositories for you. The most explicit way to do that is to use the standard Spring Data `@Enable*Repositories` and tell it the location of your `Repository` interfaces (where '*' is 'Jpa' or 'Mongo' or both).

There are also flags `spring.data.*.repositories.enabled` that you can use to switch the auto-configured repositories on and off in external configuration. This is useful for instance in case you want to switch off the Mongo repositories and still use the auto-configured `MongoTemplate`.

The same obstacle and the same features exist for other auto-configured Spring Data repository types (Elasticsearch, Solr). Just change the names of the annotations and flags respectively.

An SQL database can be initialized in different ways depending on what your stack is. Or of course you can do it manually as long as the database is a separate process.

68.1 Initialize a database using JPA

JPA has features for DDL generation, and these can be set up to run on startup against the database. This is controlled through two external properties:

- `spring.jpa.generate-ddl` (boolean) switches the feature on and off and is vendor independent.
- `spring.jpa.hibernate.ddl-auto` (enum) is a Hibernate feature that controls the behavior in a more fine-grained way. See below for more detail.

68.2 Initialize a database using Hibernate

You can set `spring.jpa.hibernate.ddl-auto` explicitly and the standard Hibernate property values are `none`, `validate`, `update`, `create-drop`. Spring Boot chooses a default value for you based on whether it thinks your database is embedded (default `create-drop`) or not (default `none`). An embedded database is detected by looking at the `Connection` type: `hsqldb`, `h2` and `derby` are embedded, the rest are not. Be careful when switching from in-memory to a 'real' database that you don't make assumptions about the existence of the tables and data in the new platform. You either have to set `ddl-auto` explicitly, or use one of the other mechanisms to initialize the database.

In addition, a file named `import.sql` in the root of the classpath will be executed on startup. This can be useful for demos and for testing if you are careful, but probably not something you want to be on the classpath in production. It is a Hibernate feature (nothing to do with Spring).

68.3 Initialize a database using Spring JDBC

Spring JDBC has a `DataSource` initializer feature. Spring Boot enables it by default and loads SQL from the standard locations `schema.sql` and `data.sql` (in the root of the classpath). In addition Spring Boot will load the `schema-${platform}.sql` and `data-${platform}.sql` files (if present), where `platform` is the value of `spring.datasource.platform`, e.g. you might choose to set it to the vendor name of the database (`hsqldb`, `h2`, `oracle`, `mysql`, `postgresql` etc.). Spring Boot enables the failfast feature of the Spring JDBC initializer by default, so if the scripts cause exceptions the application will fail to start. The script locations can be changed by setting `spring.datasource.schema` and `spring.datasource.data`, and neither location will be processed if `spring.datasource.initialize=false`.

To disable the failfast you can set `spring.datasource.continueOnError=true`. This can be useful once an application has matured and been deployed a few times, since the scripts can act as 'poor man's migrations' — inserts that fail mean that the data is already there, so there would be no need to prevent the application from running, for instance.

If you want to use the `schema.sql` initialization in a JPA app (with Hibernate) then `ddl-auto=create-drop` will lead to errors if Hibernate tries to create the same tables. To avoid those errors set `ddl-auto` explicitly to "" (preferable) or "none". Whether or not you use `ddl-auto=create-drop` you can always use `data.sql` to initialize new data.

68.4 Initialize a Spring Batch database

If you are using Spring Batch then it comes pre-packaged with SQL initialization scripts for most popular database platforms. Spring Boot will detect your database type, and execute those scripts by default,

and in this case will switch the fail fast setting to false (errors are logged but do not prevent the application from starting). This is because the scripts are known to be reliable and generally do not contain bugs, so errors are ignorable, and ignoring them makes the scripts idempotent. You can switch off the initialization explicitly using `spring.batch.initializer.enabled=false`.

68.5 Use a higher level database migration tool

Spring Boot works fine with higher level migration tools Flyway (SQL-based) and Liquibase (XML). In general we prefer Flyway because it is easier on the eyes, and it isn't very common to need platform independence: usually only one or at most couple of platforms is needed.

Execute Flyway database migrations on startup

To automatically run Flyway database migrations on startup, add the `org.flywaydb:flyway-core` to your classpath.

The migrations are scripts in the form `V<VERSION>__<NAME>.sql` (with `<VERSION>` an underscore-separated version, e.g. '1' or '2_1'). By default they live in a folder `classpath:db/migration` but you can modify that using `flyway.locations` (a list). See the Flyway class from flyway-core for details of available settings like schemas etc. In addition Spring Boot provides a small set of properties in `FlywayProperties` that can be used to disable the migrations, or switch off the location checking.

By default Flyway will autowire the (`@Primary`) `DataSource` in your context and use that for migrations. If you like to use a different `DataSource` you can create one and mark its `@Bean` as `@FlywayDataSource` - if you do that remember to create another one and mark it as `@Primary` if you want two data sources. Or you can use Flyway's native `DataSource` by setting `flyway.[url,user,password]` in external properties.

There is a Flyway sample so you can see how to set things up.

Execute Liquibase database migrations on startup

To automatically run Liquibase database migrations on startup, add the `org.liquibase:liquibase-core` to your classpath.

The master change log is by default read from `db/changelog/db.changelog-master.yaml` but can be set using `liquibase.change-log`. See `LiquibaseProperties` for details of available settings like contexts, default schema etc.

There is a Liquibase sample so you can see how to set things up.

69.1 Execute Spring Batch jobs on startup

Spring Batch auto configuration is enabled by adding `@EnableBatchProcessing` (from Spring Batch) somewhere in your context.

By default it executes **all** `Jobs` in the application context on startup (see JobLauncherCommandLineRunner for details). You can narrow down to a specific job or jobs by specifying `spring.batch.job.names` (comma-separated job name patterns).

If the application context includes a `JobRegistry` then the jobs in `spring.batch.job.names` are looked up in the registry instead of being autowired from the context. This is a common pattern with more complex systems where multiple jobs are defined in child contexts and registered centrally.

See BatchAutoConfiguration and @EnableBatchProcessing for more details.

70.1 Change the HTTP port or address of the actuator endpoints

In a standalone application the Actuator HTTP port defaults to the same as the main HTTP port. To make the application listen on a different port set the external property `management.port`. To listen on a completely different network address (e.g. if you have an internal network for management and an external one for user applications) you can also set `management.address` to a valid IP address that the server is able to bind to.

For more detail look at the `ManagementServerProperties` source code and *Section 41.3, "Customizing the management server port"* in the 'Production-ready features' section.

70.2 Customize the 'whitelabel' error page

Spring Boot installs a 'whitelabel' error page that you will see in browser client if you encounter a server error (machine clients consuming JSON and other media types should see a sensible response with the right error code). To switch it off you can set `error.whitelabel.enabled=false`, but normally in addition or alternatively to that you will want to add your own error page replacing the whitelabel one. Exactly how you do this depends on the templating technology that you are using. For example, if you are using Thymeleaf you would add an `error.html` template and if you are using FreeMarker you would add an `error.ftl` template. In general what you need is a `View` that resolves with a name of `error`, and/or a `@Controller` that handles the `/error` path. Unless you replaced some of the default configuration you should find a `BeanNameViewResolver` in your `ApplicationContext` so a `@Bean` with id `error` would be a simple way of doing that. Look at `ErrorMvcAutoConfiguration` for more options.

See also the section on Error Handling for details of how to register handlers in the servlet container.

71.1 Switch off the Spring Boot security configuration

If you define a `@Configuration` with `@EnableWebSecurity` anywhere in your application it will switch off the default webapp security settings in Spring Boot. To tweak the defaults try setting properties in `security.*` (see `SecurityProperties` for details of available settings) and `SECURITY` section of Common application properties.

71.2 Change the AuthenticationManager and add user accounts

If you provide a `@Bean` of type `AuthenticationManager` the default one will not be created, so you have the full feature set of Spring Security available (e.g. various authentication options).

Spring Security also provides a convenient `AuthenticationManagerBuilder` which can be used to build an `AuthenticationManager` with common options. The recommended way to use this in a webapp is to inject it into a void method in a `WebSecurityConfigurerAdapter`, e.g.

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
            auth.inMemoryAuthentication()
                .withUser("barry").password("password").roles("USER"); // ... etc.
    }

    // ... other stuff for application security

}
```

You will get the best results if you put this in a nested class, or a standalone class (i.e. not mixed in with a lot of other `@Beans` that might be allowed to influence the order of instantiation). The secure web sample is a useful template to follow.

If you experience instantiation issues (e.g. using JDBC or JPA for the user detail store) it might be worth extracting the `AuthenticationManagerBuilder` callback into a `GlobalAuthenticationConfigurerAdapter` (in the `init()` method so it happens before the authentication manager is needed elsewhere), e.g.

```
@Configuration
public class AuthenticationManagerConfiguration extends

    GlobalAuthenticationConfigurerAdapter {
    @Override
    public void init(AuthenticationManagerBuilder auth) {
        auth.inMemoryAuthentication() // ... etc.
    }

}
```

71.3 Enable HTTPS when running behind a proxy server

Ensuring that all your main endpoints are only available over HTTPS is an important chore for any application. If you are using Tomcat as a servlet container, then Spring Boot will add Tomcat's own `RemoteIpValve` automatically if it detects some environment settings, and you should be able to rely on the `HttpServletRequest` to report whether it is secure or not (even downstream of a proxy

server that handles the real SSL termination). The standard behavior is determined by the presence or absence of certain request headers (`x-forwarded-for` and `x-forwarded-proto`), whose names are conventional, so it should work with most front end proxies. You can switch on the valve by adding some entries to `application.properties`, e.g.

```
server.tomcat.remote_ip_header=x-forwarded-for
server.tomcat.protocol_header=x-forwarded-proto
```

(The presence of either of those properties will switch on the valve. Or you can add the `RemoteIpValve` yourself by adding a `TomcatEmbeddedServletContainerFactory` bean.)

Spring Security can also be configured to require a secure channel for all (or some requests). To switch that on in a Spring Boot application you just need to set `security.require_ssl` to `true` in `application.properties`.

72.1 Reload static content

There are several options for hot reloading. Running in an IDE (especially with debugging on) is a good way to do development (all modern IDEs allow reloading of static resources and usually also hot-swapping of Java class changes). The Maven and Gradle plugins also support running from the command line with reloading of static files. You can use that with an external css/js compiler process if you are writing that code with higher level tools.

72.2 Reload Thymeleaf templates without restarting the container

If you are using Thymeleaf, then set `spring.thymeleaf.cache` to `false`. See `ThymeleafAutoConfiguration` for other Thymeleaf customization options.

72.3 Reload FreeMarker templates without restarting the container

If you are using FreeMarker, then set `spring.freemarker.cache` to `false`. See `FreeMarkerAutoConfiguration` for other FreeMarker customization options.

72.4 Reload Groovy templates without restarting the container

If you are using Groovy templates, then set `spring.groovy.template.cache` to `false`. See `GroovyTemplateAutoConfiguration` for other Groovy customization options.

72.5 Reload Velocity templates without restarting the container

If you are using Velocity, then set `spring.velocity.cache` to `false`. See `VelocityAutoConfiguration` for other Velocity customization options.

72.6 Reload Java classes without restarting the container

Modern IDEs (Eclipse, IDEA, etc.) all support hot swapping of bytecode, so if you make a change that doesn't affect class or method signatures it should reload cleanly with no side effects.

Spring Loaded goes a little further in that it can reload class definitions with changes in the method signatures. With some customization it can force an `ApplicationContext` to refresh itself (but there is no general mechanism to ensure that would be safe for a running application anyway, so it would only ever be a development time trick probably).

Configuring Spring Loaded for use with Maven

To use Spring Loaded with the Maven command line, just add it as a dependency in the Spring Boot plugin declaration, e.g.

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
```

```
            <artifactId>springloaded</artifactId>
            <version>1.2.0.RELEASE</version>
        </dependency>
    </dependencies>
</plugin>
```

This normally works pretty well with Eclipse and IntelliJ as long as they have their build configuration aligned with the Maven defaults (Eclipse m2e does this out of the box).

Configuring Spring Loaded for use with Gradle and IntelliJ

You need to jump through a few hoops if you want to use Spring Loaded in combination with Gradle and IntelliJ. By default, IntelliJ will compile classes into a different location than Gradle, causing Spring Loaded monitoring to fail.

To configure IntelliJ correctly you can use the `idea` Gradle plugin:

```
buildscript {
    repositories { jcenter() }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-plugin:1.2.0.RELEASE"
        classpath 'org.springframework:springloaded:1.2.0.RELEASE'
    }
}

apply plugin: 'idea'

idea {
    module {
        inheritOutputDirs = false
        outputDir = file("$buildDir/classes/main/")
    }
}

// ...
```

> **Note**
>
> IntelliJ must be configured to use the same Java version as the command line Gradle task and `springloaded` **must** be included as a `buildscript` dependency.

You can also additionally enable ‘Make Project Automatically’ inside Intellij to automatically compile your code whenever a file is saved.

73.1 Customize dependency versions with Maven

If you use a Maven build that inherits directly or indirectly from `spring-boot-dependencies` (for instance `spring-boot-starter-parent`) but you want to override a specific third-party dependency you can add appropriate `<properties>` elements. Browse the `spring-boot-dependencies` POM for a complete list of properties. For example, to pick a different `slf4j` version you would add the following:

```
<properties>
    <slf4j.version>1.7.5<slf4j.version>
</properties>
```

> **Note**
>
> This only works if your Maven project inherits (directly or indirectly) from `spring-boot-dependencies`. If you have added `spring-boot-dependencies` in your own `dependencyManagement` section with `<scope>import</scope>` you have to redefine the artifact yourself instead of overriding the property .

> **Warning**
>
> Each Spring Boot release is designed and tested against a specific set of third-party dependencies. Overriding versions may cause compatibility issues.

73.2 Create an executable JAR with Maven

The `spring-boot-maven-plugin` can be used to create an executable 'fat' JAR. If you are using the `spring-boot-starter-parent` POM you can simply declare the plugin and your jars will be repackaged:

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

If you are not using the parent POM you can still use the plugin, however, you must additionally add an `<executions>` section:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>1.2.0.RELEASE</version>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

See the plugin documentation for full usage details.

73.3 Create an additional executable JAR

If you want to use your project as a library jar for other projects to depend on, and in addition have an executable (e.g. demo) version of it, you will want to configure the build in a slightly different way.

For Maven the normal JAR plugin and the Spring Boot plugin both have a ‘classifier’ configuration that you can add to create an additional JAR. Example (using the Spring Boot Starter Parent to manage the plugin versions and other configuration defaults):

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

Two jars are produced, the default one, and an executable one using the Boot plugin with classifier ‘exec’.

For Gradle users the steps are similar. Example:

```
bootRepackage  {
    classifier = 'exec'
}
```

73.4 Extract specific libraries when an executable jar runs

Most nested libraries in an executable jar do not need to be unpacked in order to run, however, certain libraries can have problems. For example, JRuby includes its own nested jar support which assumes that the `jruby-complete.jar` is always directly available as a file in its own right.

To deal with any problematic libraries, you can flag that specific nested jars should be automatically unpacked to the ‘temp folder’ when the executable jar first runs.

For example, to indicate that JRuby should be flagged for unpack using the Maven Plugin you would add the following configuration:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <requiresUnpack>
                    <dependency>
                        <groupId>org.jruby</groupId>
                        <artifactId>jruby-complete</artifactId>
                    </dependency>
                </requiresUnpack>
            </configuration>
        </plugin>
    </plugins>
</build>
```

And to do that same with Gradle:

```
springBoot  {
    requiresUnpack = ['org.jruby:jruby-complete']
}
```

73.5 Create a non-executable JAR with exclusions

Often if you have an executable and a non-executable jar as build products, the executable version will have additional configuration files that are not needed in a library jar. E.g. the `application.yml` configuration file might excluded from the non-executable JAR.

Here's how to do that in Maven:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <classifier>exec</classifier>
            </configuration>
        </plugin>
        <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <executions>
                <execution>
                    <id>exec</id>
                    <phase>package</phase>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                    <configuration>
                        <classifier>exec</classifier>
                    </configuration>
                </execution>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                    <configuration>
                        <!-- Need this to ensure application.yml is excluded -->
                        <forceCreation>true</forceCreation>
                        <excludes>
                            <exclude>application.yml</exclude>
                        </excludes>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

In Gradle you can create a new JAR archive with standard task DSL features, and then have the `bootRepackage` task depend on that one using its `withJarTask` property:

```
jar {
    baseName = 'spring-boot-sample-profile'
    version =  '0.0.0'
    excludes = ['**/application.yml']
}

task('execJar', type:Jar, dependsOn: 'jar') {
    baseName = 'spring-boot-sample-profile'
    version =  '0.0.0'
```

```
    classifier = 'exec'
    from sourceSets.main.output
}

bootRepackage  {
    withJarTask = tasks['execJar']
}
```

73.6 Remote debug a Spring Boot application started with Maven

To attach a remote debugger to a Spring Boot application started with Maven you can use the `jvmArguments` property of the maven plugin.

Check this example for more details.

73.7 Remote debug a Spring Boot application started with Gradle

To attach a remote debugger to a Spring Boot application started with Gradle you can use the `applicationDefaultJvmArgs` in `build.gradle` or `--debug-jvm` command line option.

`build.gradle`:

```
applicationDefaultJvmArgs = [
    "-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005"
]
```

Command line:

```
$ gradle run --debug-jvm
```

Check Gradle Application Plugin for more details.

73.8 Build an executable archive with Ant

To build with Ant you need to grab dependencies, compile and then create a jar or war archive as normal. To make it executable:

1. Use the appropriate launcher as a `Main-Class`, e.g. `JarLauncher` for a jar file, and specify the other properties it needs as manifest entries, principally a `Start-Class`.
2. Add the runtime dependencies in a nested 'lib' directory (for a jar) and the `provided` (embedded container) dependencies in a nested `lib-provided` directory. Remember **not** to compress the entries in the archive.
3. Add the `spring-boot-loader` classes at the root of the archive (so the `Main-Class` is available).

Example:

```
<target name="build" depends="compile">
    <copy todir="target/classes/lib">
        <fileset dir="lib/runtime" />
    </copy>
    <jar destfile="target/spring-boot-sample-actuator-${spring-boot.version}.jar" compress="false">
        <fileset dir="target/classes" />
        <fileset dir="src/main/resources" />
```

```
        <zipfileset src="lib/loader/spring-boot-loader-jar-${spring-boot.version}.jar" />
        <manifest>
            <attribute name="Main-Class" value="org.springframework.boot.loader.JarLauncher" />
            <attribute name="Start-Class" value="${start-class}" />
        </manifest>
    </jar>
</target>
```

The Actuator Sample has a `build.xml` that should work if you run it with

```
$ ant -lib <path_to>/ivy-2.2.jar
```

after which you can run the application with

```
$ java -jar target/*.jar
```

74.1 Create a deployable war file

Use the `SpringBootServletInitializer` base class, which is picked up by Spring's Servlet 3.0 support on deployment. Add an extension of that to your project and build a war file as normal. For more detail, see the 'Converting a jar Project to a war' guide on the spring.io website and the sample below.

The war file can also be executable if you use the Spring Boot build tools. In that case the embedded container classes (to launch Tomcat for instance) have to be added to the war in a `lib-provided` directory. The tools will take care of that as long as the dependencies are marked as 'provided' in Maven or Gradle. Here's a Maven example in the Boot Samples.

74.2 Create a deployable war file for older servlet containers

Older Servlet containers don't have support for the `ServletContextInitializer` bootstrap process used in Servlet 3.0. You can still use Spring and Spring Boot in these containers but you are going to need to add a `web.xml` to your application and configure it to load an `ApplicationContext` via a `DispatcherServlet`.

74.3 Convert an existing application to Spring Boot

For a non-web application it should be easy (throw away the code that creates your `ApplicationContext` and replace it with calls to `SpringApplication` or `SpringApplicationBuilder`). Spring MVC web applications are generally amenable to first creating a deployable war application, and then migrating it later to an executable war and/or jar. Useful reading is in the Getting Started Guide on Converting a jar to a war.

Create a deployable war by extending `SpringBootServletInitializer` (e.g. in a class called `Application`), and add the Spring Boot `@EnableAutoConfiguration` annotation. Example:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        // Customize the application or call application.sources(...) to add sources
        // Since our example is itself a @Configuration class we actually don't
        // need to override this method.
        return application;
    }

}
```

Remember that whatever you put in the `sources` is just a Spring `ApplicationContext` and normally anything that already works should work here. There might be some beans you can remove later and let Spring Boot provide its own defaults for them, but it should be possible to get something working first.

Static resources can be moved to `/public` (or `/static` or `/resources` or `/META-INF/resources`) in the classpath root. Same for `messages.properties` (Spring Boot detects this automatically in the root of the classpath).

Vanilla usage of Spring `DispatcherServlet` and Spring Security should require no further changes. If you have other features in your application, using other servlets or filters for instance, then you may need to add some configuration to your `Application` context, replacing those elements from the `web.xml` as follows:

- A `@Bean` of type `Servlet` or `ServletRegistrationBean` installs that bean in the container as if it was a `<servlet/>` and `<servlet-mapping/>` in `web.xml`.
- A `@Bean` of type `Filter` or `FilterRegistrationBean` behaves similarly (like a `<filter/>` and `<filter-mapping/>`.
- An `ApplicationContext` in an XML file can be added to an `@Import` in your `Application`. Or simple cases where annotation configuration is heavily used already can be recreated in a few lines as `@Bean` definitions.

Once the war is working we make it executable by adding a `main` method to our `Application`, e.g.

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

Applications can fall into more than one category:

- Servlet 3.0+ applications with no `web.xml`.
- Applications with a `web.xml`.
- Applications with a context hierarchy.
- Applications without a context hierarchy.

All of these should be amenable to translation, but each might require slightly different tricks.

Servlet 3.0+ applications might translate pretty easily if they already use the Spring Servlet 3.0+ initializer support classes. Normally all the code from an existing `WebApplicationInitializer` can be moved into a `SpringBootServletInitializer`. If your existing application has more than one `ApplicationContext` (e.g. if it uses `AbstractDispatcherServletInitializer`) then you might be able to squash all your context sources into a single `SpringApplication`. The main complication you might encounter is if that doesn't work and you need to maintain the context hierarchy. See the entry on building a hierarchy for examples. An existing parent context that contains web-specific features will usually need to be broken up so that all the `ServletContextAware` components are in the child context.

Applications that are not already Spring applications might be convertible to a Spring Boot application, and the guidance above might help, but your mileage may vary.

74.4 Deploying a WAR to Weblogic

To deploy a Spring Boot application to Weblogic you must ensure that your servlet initializer **directly** implements `WebApplicationInitializer` (even if you extend from a base class that already implements it).

A typical initializer for Weblogic would be something like this:

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.web.SpringBootServletInitializer;
import org.springframework.web.WebApplicationInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer implements WebApplicationInitializer {

}
```

If you use logback, you will also need to tell Weblogic to prefer the packaged version rather than the version that pre-installed with the server. You can do this by adding a `WEB-INF/weblogic.xml` file with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<wls:weblogic-web-app
 xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-web-app"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
  http://xmlns.oracle.com/weblogic/weblogic-web-app
  http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd">
 <wls:container-descriptor>
  <wls:prefer-application-packages>
   <wls:package-name>org.slf4j</wls:package-name>
  </wls:prefer-application-packages>
 </wls:container-descriptor>
</wls:weblogic-web-app>
```

74.5 Deploying a WAR in an Old (Servlet 2.5) Container

Spring Boot uses Servet 3.0 APIs to initialize the `ServletContext` (register `Servlets` etc.) so you can't use the same application out of the box in a Servlet 2.5 container. It **is** however possible to run a Spring Boot application on an older container with some special tools. If you include `org.springframework.boot:spring-boot-legacy` as a dependency (maintained separately to the core of Spring Boot and currently available at 1.0.0.RELEASE), all you should need to do is create a `web.xml` and declare a context listener to create the application context and your filters and servlets. The context listener is a special purpose one for Spring Boot, but the rest of it is normal for a Spring application in Servlet 2.5. Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd">

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>demo.Application</param-value>
    </context-param>

    <listener>
        <listener-class>org.springframework.boot.legacy.context.web.SpringBootContextLoaderListener</
listener-class>
    </listener>

    <filter>
        <filter-name>metricFilter</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filte>r

    <filter-mapping>
        <filter-name>metricFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextAttribute</param-name>
            <param-value>org.springframework.web.context.WebApplicationContext.ROOT</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

```
    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

In this example we are using a single application context (the one created by the context listener) and attaching it to the `DispatcherServlet` using an init parameter. This is normal in a Spring Boot application (you normally only have one application context).

Part X. Appendices

Appendix A. Common application properties

Various properties can be specified inside your `application.properties`/`application.yml` file or as command line switches. This section provides a list common Spring Boot properties and references to the underlying classes that consume them.

> **Note**
>
> Property contributions can come from additional jar files on your classpath so you should not consider this an exhaustive list. It is also perfectly legit to define your own properties.

> **Warning**
>
> This sample file is meant as a guide only. Do **not** copy/paste the entire content into your application; rather pick only the properties that you need.

```
# ===================================================================
# COMMON SPRING BOOT PROPERTIES
#
# This sample file is provided as a guideline. Do NOT copy it in its
# entirety to your own application.               ^^^
# ===================================================================

# ----------------------------------------
# CORE PROPERTIES
# ----------------------------------------

# SPRING CONFIG (ConfigFileApplicationListener)
spring.config.name= # config file name (default to 'application')
spring.config.location= # location of config file

# PROFILES
spring.profiles.active= # comma list of active profiles
spring.profiles.include= # unconditionally activate the specified comma separated profiles

# APPLICATION SETTINGS (SpringApplication)
spring.main.sources=
spring.main.web-environment= # detect by default
spring.main.show-banner=true
spring.main....= # see class for all properties

# LOGGING
logging.path=/var/logs
logging.file=myapp.log
logging.config= # location of config file (default classpath:logback.xml for logback)
logging.level.*= # levels for loggers, e.g. "logging.level.org.springframework=DEBUG" (TRACE, DEBUG,
 INFO, WARN, ERROR, FATAL, OFF)

# IDENTITY (ContextIdApplicationContextInitializer)
spring.application.name=
spring.application.index=

# EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.port=8080
server.address= # bind to a specific NIC
server.session-timeout= # session timeout in seconds
server.context-parameters.*= # Servlet context init parameters, e.g. server.context-parameters.a=alpha
server.context-path= # the context path, defaults to '/'
server.servlet-path= # the servlet path, defaults to '/'
server.ssl.client-auth= # want or need
```

```
server.ssl.key-alias=
server.ssl.ciphers= # supported SSL ciphers
server.ssl.key-password=
server.ssl.key-store=
server.ssl.key-store-password=
server.ssl.key-store-provider=
server.ssl.key-store-type=
server.ssl.protocol=TLS
server.ssl.trust-store=
server.ssl.trust-store-password=
server.ssl.trust-store-provider=
server.ssl.trust-store-type=
server.tomcat.access-log-pattern= # log pattern of the access log
server.tomcat.access-log-enabled=false # is access logging enabled
server.tomcat.internal-proxies=10\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\
       192\\.168\\.\\d{1,3}\\.\\d{1,3}|\\
       169\\.254\\.\\d{1,3}\\.\\d{1,3}|\\
       127\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3} # regular expression matching trusted IP addresses
server.tomcat.protocol-header=x-forwarded-proto # front end proxy forward header
server.tomcat.port-header= # front end proxy port header
server.tomcat.remote-ip-header=x-forwarded-for
server.tomcat.basedir=/tmp # base dir (usually not needed, defaults to tmp)
server.tomcat.background-processor-delay=30; # in seconds
server.tomcat.max-http-header-size= # maximum size in bytes of the HTTP message header
server.tomcat.max-threads = 0 # number of threads in protocol handler
server.tomcat.uri-encoding = UTF-8 # character encoding to use for URL decoding

# SPRING MVC (WebMvcProperties)
spring.mvc.locale= # set fixed locale, e.g. en_UK
spring.mvc.date-format= # set fixed date format, e.g. dd/MM/yyyy
spring.mvc.message-codes-resolver-format= # PREFIX_ERROR_CODE / POSTFIX_ERROR_CODE
spring.mvc.ignore-default-model-on-redirect=true # If the the content of the "default" model should be
 ignored redirects
spring.view.prefix= # MVC view prefix
spring.view.suffix= # ... and suffix
spring.resources.cache-period= # cache timeouts in headers sent to browser
spring.resources.add-mappings=true # if default mappings should be added

# HTTP encoding (HttpEncodingProperties)
spring.http.encoding.charset=UTF-8 # the encoding of HTTP requests/responses
spring.http.encoding.enabled=true # enable http encoding support
spring.http.encoding.force=true # force the configured encoding

# JACKSON (JacksonProperties)
spring.jackson.date-format= # Date format string (e.g. yyyy-MM-dd HH:mm:ss), or a fully-qualified date
 format class name (e.g. com.fasterxml.jackson.databind.util.ISO8601DateFormat)
spring.jackson.property-naming-strategy= # One of the constants on Jackson's PropertyNamingStrategy
 (e.g. CAMEL_CASE_TO_LOWER_CASE_WITH_UNDERSCORES) or the fully-qualified class name of a
 PropertyNamingStrategy subclass
spring.jackson.deserialization.*= # see Jackson's DeserializationFeature
spring.jackson.generator.*= # see Jackson's JsonGenerator.Feature
spring.jackson.mapper.*= # see Jackson's MapperFeature
spring.jackson.parser.*= # see Jackson's JsonParser.Feature
spring.jackson.serialization.*= # see Jackson's SerializationFeature

# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.check-template-location=true
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.excluded-view-names= # comma-separated list of view names that should be excluded from
 resolution
spring.thymeleaf.view-names= # comma-separated list of view names that can be resolved
spring.thymeleaf.suffix=.html
spring.thymeleaf.mode=HTML5
spring.thymeleaf.encoding=UTF-8
spring.thymeleaf.content-type=text/html # ;charset=<encoding> is added
spring.thymeleaf.cache=true # set to false for hot refresh

# FREEMARKER (FreeMarkerAutoConfiguration)
spring.freemarker.allow-request-override=false
spring.freemarker.cache=true
spring.freemarker.check-template-location=true
```

```
spring.freemarker.charset=UTF-8
spring.freemarker.content-type=text/html
spring.freemarker.expose-request-attributes=false
spring.freemarker.expose-session-attributes=false
spring.freemarker.expose-spring-macro-helpers=false
spring.freemarker.prefix=
spring.freemarker.request-context-attribute=
spring.freemarker.settings.*=
spring.freemarker.suffix=.ftl
spring.freemarker.template-loader-path=classpath:/templates/ # comma-separated list
spring.freemarker.view-names= # whitelist of view names that can be resolved

# GROOVY TEMPLATES (GroovyTemplateAutoConfiguration)
spring.groovy.template.cache=true
spring.groovy.template.charset=UTF-8
spring.groovy.template.configuration.*= # See Groovy's TemplateConfiguration
spring.groovy.template.content-type=text/html
spring.groovy.template.prefix=classpath:/templates/
spring.groovy.template.suffix=.tpl
spring.groovy.template.view-names= # whitelist of view names that can be resolved

# VELOCITY TEMPLATES (VelocityAutoConfiguration)
spring.velocity.allow-request-override=false
spring.velocity.cache=true
spring.velocity.check-template-location=true
spring.velocity.charset=UTF-8
spring.velocity.content-type=text/html
spring.velocity.date-tool-attribute=
spring.velocity.expose-request-attributes=false
spring.velocity.expose-session-attributes=false
spring.velocity.expose-spring-macro-helpers=false
spring.velocity.number-tool-attribute=
spring.velocity.prefer-file-system-access=true # prefer file system access for template loading
spring.velocity.prefix=
spring.velocity.properties.*=
spring.velocity.request-context-attribute=
spring.velocity.resource-loader-path=classpath:/templates/
spring.velocity.suffix=.vm
spring.velocity.toolbox-config-location= # velocity Toolbox config location, for example "/WEB-INF/
toolbox.xml"
spring.velocity.view-names= # whitelist of view names that can be resolved

# JERSEY (JerseyProperties)
spring.jersey.type=servlet # servlet or filter
spring.jersey.init= # init params
spring.jersey.filter.order=

# INTERNATIONALIZATION (MessageSourceAutoConfiguration)
spring.messages.basename=messages
spring.messages.cache-seconds=-1
spring.messages.encoding=UTF-8


# SECURITY (SecurityProperties)
security.user.name=user # login username
security.user.password= # login password
security.user.role=USER # role assigned to the user
security.require-ssl=false # advanced settings ...
security.enable-csrf=false
security.basic.enabled=true
security.basic.realm=Spring
security.basic.path= # /**
security.filter-order=0
security.headers.xss=false
security.headers.cache=false
security.headers.frame=false
security.headers.content-type=false
security.headers.hsts=all # none / domain / all
security.sessions=stateless # always / never / if_required / stateless
security.ignored=false
```

```
# DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.name= # name of the data source
spring.datasource.initialize=true # populate using data.sql
spring.datasource.schema= # a schema (DDL) script resource reference
spring.datasource.data= # a data (DML) script resource reference
spring.datasource.sql-script-encoding= # a charset for reading SQL scripts
spring.datasource.platform= # the platform to use in the schema resource (schema-${platform}.sql)
spring.datasource.continue-on-error=false # continue even if can't be initialized
spring.datasource.separator=; # statement separator in SQL initialization scripts
spring.datasource.driver-class-name= # JDBC Settings...
spring.datasource.url=
spring.datasource.username=
spring.datasource.password=
spring.datasource.jndi-name # For JNDI lookup (class, url, username & password are ignored when set)
spring.datasource.max-active=100 # Advanced configuration...
spring.datasource.max-idle=8
spring.datasource.min-idle=8
spring.datasource.initial-size=10
spring.datasource.validation-query=
spring.datasource.test-on-borrow=false
spring.datasource.test-on-return=false
spring.datasource.test-while-idle=
spring.datasource.time-between-eviction-runs-millis=
spring.datasource.min-evictable-idle-time-millis=
spring.datasource.max-wait=
spring.datasource.jmx-enabled=false # Export JMX MBeans (if supported)

# DATASOURCE (PersistenceExceptionTranslationAutoConfiguration
spring.dao.exceptiontranslation.enabled=true

# MONGODB (MongoProperties)
spring.data.mongodb.host= # the db host
spring.data.mongodb.port=27017 # the connection port (defaults to 27107)
spring.data.mongodb.uri=mongodb://localhost/test # connection URL
spring.data.mongodb.database=
spring.data.mongodb.authentication-database=
spring.data.mongodb.grid-fs-database=
spring.data.mongodb.username=
spring.data.mongodb.password=
spring.data.mongodb.repositories.enabled=true # if spring data repository support is enabled

# JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.jpa.properties.*= # properties to set on the JPA connection
spring.jpa.open-in-view=true
spring.jpa.show-sql=true
spring.jpa.database-platform=
spring.jpa.database=
spring.jpa.generate-ddl=false # ignored by Hibernate, might be useful for other vendors
spring.jpa.hibernate.naming-strategy= # naming classname
spring.jpa.hibernate.ddl-auto= # defaults to create-drop for embedded dbs
spring.data.jpa.repositories.enabled=true # if spring data repository support is enabled

# JTA (JtaAutoConfiguration)
spring.jta.log-dir= # transaction log dir
spring.jta.*= # technology specific configuration

# SOLR (SolrProperties})
spring.data.solr.host=http://127.0.0.1:8983/solr
spring.data.solr.zk-host=
spring.data.solr.repositories.enabled=true # if spring data repository support is enabled

# ELASTICSEARCH (ElasticsearchProperties})
spring.data.elasticsearch.cluster-name= # The cluster name (defaults to elasticsearch)
spring.data.elasticsearch.cluster-nodes= # The address(es) of the server node (comma-separated; if not
 specified starts a client node)
spring.data.elasticsearch.repositories.enabled=true # if spring data repository support is enabled

# DATA RESET (RepositoryRestConfiguration})
spring.data.rest.base-uri= # base URI against which the exporter should calculate its links

# FLYWAY (FlywayProperties)
```

```
flyway.check-location=false # check that migration scripts location exists
flyway.locations=classpath:db/migration # locations of migrations scripts
flyway.schemas= # schemas to update
flyway.init-version= 1 # version to start migration
flyway.init-sqls= # SQL statements to execute to initialize a connection immediately after obtaining it
flyway.sql-migration-prefix=V
flyway.sql-migration-suffix=.sql
flyway.enabled=true
flyway.url= # JDBC url if you want Flyway to create its own DataSource
flyway.user= # JDBC username if you want Flyway to create its own DataSource
flyway.password= # JDBC password if you want Flyway to create its own DataSource

# LIQUIBASE (LiquibaseProperties)
liquibase.change-log=classpath:/db/changelog/db.changelog-master.yaml
liquibase.check-change-log-location=true # check the change log location exists
liquibase.contexts= # runtime contexts to use
liquibase.default-schema= # default database schema to use
liquibase.drop-first=false
liquibase.enabled=true
liquibase.url= # specific JDBC url (if not set the default datasource is used)
liquibase.user= # user name for liquibase.url
liquibase.password= # password for liquibase.url

# JMX
spring.jmx.enabled=true # Expose MBeans from Spring

# RABBIT (RabbitProperties)
spring.rabbitmq.host= # connection host
spring.rabbitmq.port= # connection port
spring.rabbitmq.addresses= # connection addresses (e.g. myhost:9999,otherhost:1111)
spring.rabbitmq.username= # login user
spring.rabbitmq.password= # login password
spring.rabbitmq.virtual-host=
spring.rabbitmq.dynamic=

# REDIS (RedisProperties)
spring.redis.database= # database name
spring.redis.host=localhost # server host
spring.redis.password= # server password
spring.redis.port=6379 # connection port
spring.redis.pool.max-idle=8 # pool settings ...
spring.redis.pool.min-idle=0
spring.redis.pool.max-active=8
spring.redis.pool.max-wait=-1
spring.redis.sentinel.master= # name of Redis server
spring.redis.sentinel.nodes= # comma-separated list of host:port pairs

# ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url=tcp://localhost:61616 # connection URL
spring.activemq.user=
spring.activemq.password=
spring.activemq.in-memory=true # broker kind to create if no broker-url is specified
spring.activemq.pooled=false

# HornetQ (HornetQProperties)
spring.hornetq.mode= # connection mode (native, embedded)
spring.hornetq.host=localhost # hornetQ host (native mode)
spring.hornetq.port=5445 # hornetQ port (native mode)
spring.hornetq.embedded.enabled=true # if the embedded server is enabled (needs hornetq-jms-server.jar)
spring.hornetq.embedded.server-id= # auto-generated id of the embedded server (integer)
spring.hornetq.embedded.persistent=false # message persistence
spring.hornetq.embedded.data-directory= # location of data content (when persistence is enabled)
spring.hornetq.embedded.queues= # comma-separated queues to create on startup
spring.hornetq.embedded.topics= # comma-separated topics to create on startup
spring.hornetq.embedded.cluster-password= # customer password (randomly generated by default)

# JMS (JmsProperties)
spring.jms.jndi-name= # JNDI location of a JMS ConnectionFactory
spring.jms.pub-sub-domain= # false for queue (default), true for topic

# Email (MailProperties)
```

```
spring.mail.host=smtp.acme.org # mail server host
spring.mail.port= # mail server port
spring.mail.username=
spring.mail.password=
spring.mail.default-encoding=UTF-8 # encoding to use for MimeMessages
spring.mail.properties.*= # properties to set on the JavaMail session

# SPRING BATCH (BatchDatabaseInitializer)
spring.batch.job.names=job1,job2
spring.batch.job.enabled=true
spring.batch.initializer.enabled=true
spring.batch.schema= # batch schema to load

# AOP
spring.aop.auto=
spring.aop.proxy-target-class=

# FILE ENCODING (FileEncodingApplicationListener)
spring.mandatory-file-encoding=false

# SPRING SOCIAL (SocialWebAutoConfiguration)
spring.social.auto-connection-views=true # Set to true for default connection views or false if you
 provide your own

# SPRING SOCIAL FACEBOOK (FacebookAutoConfiguration)
spring.social.facebook.app-id= # your application's Facebook App ID
spring.social.facebook.app-secret= # your application's Facebook App Secret

# SPRING SOCIAL LINKEDIN (LinkedInAutoConfiguration)
spring.social.linkedin.app-id= # your application's LinkedIn App ID
spring.social.linkedin.app-secret= # your application's LinkedIn App Secret

# SPRING SOCIAL TWITTER (TwitterAutoConfiguration)
spring.social.twitter.app-id= # your application's Twitter App ID
spring.social.twitter.app-secret= # your application's Twitter App Secret

# SPRING MOBILE SITE PREFERENCE (SitePreferenceAutoConfiguration)
spring.mobile.sitepreference.enabled=true # enabled by default

# SPRING MOBILE DEVICE VIEWS (DeviceDelegatingViewResolverAutoConfiguration)
spring.mobile.devicedelegatingviewresolver.enabled=true # disabled by default
spring.mobile.devicedelegatingviewresolver.normal-prefix=
spring.mobile.devicedelegatingviewresolver.normal-suffix=
spring.mobile.devicedelegatingviewresolver.mobile-prefix=mobile/
spring.mobile.devicedelegatingviewresolver.mobile-suffix=
spring.mobile.devicedelegatingviewresolver.tablet-prefix=tablet/
spring.mobile.devicedelegatingviewresolver.tablet-suffix=

# ----------------------------------------
# ACTUATOR PROPERTIES
# ----------------------------------------

# MANAGEMENT HTTP SERVER (ManagementServerProperties)
management.port= # defaults to 'server.port'
management.address= # bind to a specific NIC
management.context-path= # default to '/'
management.add-application-context-header= # default to true
management.security.enabled=true # enable security
management.security.role=ADMIN # role required to access the management endpoint
management.security.sessions=stateless # session creating policy to use (always, never, if_required,
 stateless)

# PID FILE (ApplicationPidFileWriter)
spring.pidfile= # Location of the PID file to write

# ENDPOINTS (AbstractEndpoint subclasses)
endpoints.autoconfig.id=autoconfig
endpoints.autoconfig.sensitive=true
endpoints.autoconfig.enabled=true
endpoints.beans.id=beans
endpoints.beans.sensitive=true
```

```
endpoints.beans.enabled=true
endpoints.configprops.id=configprops
endpoints.configprops.sensitive=true
endpoints.configprops.enabled=true
endpoints.configprops.keys-to-sanitize=password,secret,key # suffix or regex
endpoints.dump.id=dump
endpoints.dump.sensitive=true
endpoints.dump.enabled=true
endpoints.env.id=env
endpoints.env.sensitive=true
endpoints.env.enabled=true
endpoints.env.keys-to-sanitize=password,secret,key # suffix or regex
endpoints.health.id=health
endpoints.health.sensitive=true
endpoints.health.enabled=true
endpoints.health.mapping.*= # mapping of health statuses to HttpStatus codes
endpoints.health.time-to-live=1000
endpoints.info.id=info
endpoints.info.sensitive=false
endpoints.info.enabled=true
endpoints.mappings.enabled=true
endpoints.mappings.id=mappings
endpoints.mappings.sensitive=true
endpoints.metrics.id=metrics
endpoints.metrics.sensitive=true
endpoints.metrics.enabled=true
endpoints.shutdown.id=shutdown
endpoints.shutdown.sensitive=true
endpoints.shutdown.enabled=false
endpoints.trace.id=trace
endpoints.trace.sensitive=true
endpoints.trace.enabled=true

# HEALTH INDICATORS
management.health.db.enabled=true
management.health.diskspace.enabled=true
management.health.mongo.enabled=true
management.health.rabbit.enabled=true
management.health.redis.enabled=true
management.health.solr.enabled=true
management.health.diskspace.path=.
management.health.diskspace.threshold=10485760
management.health.status.order: DOWN, OUT_OF_SERVICE, UNKNOWN, UP

# MVC ONLY ENDPOINTS
endpoints.jolokia.path=jolokia
endpoints.jolokia.sensitive=true
endpoints.jolokia.enabled=true # when using Jolokia

# JMX ENDPOINT (EndpointMBeanExportProperties)
endpoints.jmx.enabled=true
endpoints.jmx.domain= # the JMX domain, defaults to 'org.springboot'
endpoints.jmx.unique-names=false
endpoints.jmx.static-names=

# JOLOKIA (JolokiaProperties)
jolokia.config.*= # See Jolokia manual

# REMOTE SHELL
shell.auth=simple # jaas, key, simple, spring
shell.command-refresh-interval=-1
shell.command-path-patterns= # classpath*:/commands/**, classpath*:/crash/commands/**
shell.config-path-patterns= # classpath*:/crash/*
shell.disabled-commands=jpa*,jdbc*,jndi* # comma-separated list of commands to disable
shell.disabled-plugins=false # don't expose plugins
shell.ssh.enabled= # ssh settings ...
shell.ssh.key-path=
shell.ssh.port=
shell.telnet.enabled= # telnet settings ...
shell.telnet.port=
shell.auth.jaas.domain= # authentication settings ...
```

```
shell.auth.key.path=
shell.auth.simple.user.name=
shell.auth.simple.user.password=
shell.auth.spring.roles=

# GIT INFO
spring.git.properties= # resource ref to generated git info properties file
```

Appendix B. Configuration meta-data

Spring Boot jars are shipped with meta-data files that provide details of all supported configuration properties. The files are designed to allow IDE developers to offer contextual help and "code completion" as users are working with `application.properties` or `application.yml` files.

The majority of the meta-data file is generated automatically at compile time by processing all items annotated with `@ConfigurationProperties`.

B.1 Meta-data format

Configuration meta-data files are located inside jars under `META-INF/spring-configuration-metadata.json` They use a simple JSON format with items categorized under either "groups" or "properties":

```
{"groups": [
    {
        "name": "server",
        "type": "org.springframework.boot.autoconfigure.web.ServerProperties",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    }
    ...
],"properties": [
    {
        "name": "server.port",
        "type": "java.lang.Integer",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
        "name": "server.servlet-path",
        "type": "java.lang.String",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
        "defaultValue": "/"
    }
    ...
]}
```

Each "property" is a configuration item that the user specifies with a given value. For example `server.port` and `server.servlet-path` might be specified in `application.properties` as follows:

```
server.port=9090
server.servlet-path=/home
```

The "groups" are higher level items that don't themselves specify a value, but instead provide a contextual grouping for properties. For example the `server.port` and `server.servlet-path` properties are part of the `server` group.

> **Note**
>
> It is not required that every "property" has a "group", some properties might just exist in their own right.

Group Attributes

The JSON object contained in the `groups` array can contain the following attributes:

Name	Type	Purpose
`name`	String	The full name of the group. This attribute is mandatory.
`type`	String	The class name of the data type of the group. For example, if the group was based on a class annotated with `@ConfigurationProperties` the attribute would contain the fully qualified name of that class. If it was based on a `@Bean` method, it would be the return type of that method. The attribute may be omitted if the type is not known.
`description`	String	A short description of the group that can be displayed to users. May be omitted if no description is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`sourceType`	String	The class name of the source that contributed this group. For example, if the group was based on a `@Bean` method annotated with `@ConfigurationProperties` this attribute would contain the fully qualified name of the `@Configuration` class containing the method. The attribute may be omitted if the source type is not known.
`sourceMethod`	String	The full name of the method (include parenthesis and argument types) that contributed this group. For example, the name of a `@ConfigurationProperties` annotated `@Bean` method. May be omitted if the source method is not known.

Property Attributes

The JSON object contained in the `properties` array can contain the following attributes:

Name	Type	Purpose
`name`	String	The full name of the property. Names are in lowercase dashed form (e.g. `server.servlet-path`). This attribute is mandatory.
`type`	String	The class name of the data type of the property. For example, `java.lang.String`. This attribute can be used to guide the user as to the types of values that they can enter. For consistency, the type of a primitive is specified using its wrapper counterpart, i.e. `boolean` becomes `java.lang.Boolean`. Collection types are harmonized to their interface counterpart and define the actual generic types, i.e. `java.util.HashMap<java.lang.String,java.lang.Integer>` becomes `java.util.Map<java.lang.String,java.lang.Integer>`. Note that this class may be a complex type that gets converted from a String as values are bound. May be omitted if the type is not known.

Name	Type	Purpose
`description`	String	A short description of the group that can be displayed to users. May be omitted if no description is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`sourceType`	String	The class name of the source that contributed this property. For example, if the property was from a class annotated with `@ConfigurationProperties` this attribute would contain the fully qualified name of that class. May be omitted if the source type is not known.
`defaultValue`	Object	The default value which will be used if the property is not specified. Can also be an array of value(s) if the type of the property is an array. May be omitted if the default value is not known.
`deprecated`	boolean	Specify if the property is deprecated. May be omitted if the field is not deprecated or if that information is not known.

Repeated meta-data items

It is perfectly acceptable for “property” and “group” objects with the same name to appear multiple times within a meta-data file. For example, Spring Boot binds `spring.datasource` properties to Hikari, Tomcat and DBCP classes, with each potentially offering overlap of property names. Consumers of meta-data should take care to ensure that they support such scenarios.

B.2 Generating your own meta-data using the annotation processor

You can easily generate your own configuration meta-data file from items annotated with `@ConfigurationProperties` by using the `spring-boot-configuration-processor` jar. The jar includes a Java annotation processor which is invoked as your project is compiled. To use the processor, simply include `spring-boot-configuration-processor` as an optional dependency, for example with Maven you would add:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

The annotation will pickup both classes and methods that are annotated with `@ConfigurationProperties`. The Javadoc for field values within configuration classes will be used to populate the `description` attribute.

> **Note**
>
> You should only use simple text with `@ConfigurationProperties` field Javadoc since they are not processed before being added to the JSON.

Nested properties

The annotation processor will automatically consider inner classes as nested properties. For example, the following class:

```
@ConfigurationProperties(prefix="server")
public class ServerProperties {

    private String name;

    private Host host;

    // ... getter and setters

    private static class Host {

        private String ip;

        private int port;

        // ... getter and setters

    }

}
```

Will produce meta-data information for `server.name`, `server.host.ip` and `server.host.port` properties. You can use the `@NestedConfigurationProperty` annotation on a field to indicate that a regular (non-inner) class should be treated as if it were nested.

Adding additional meta-data

Spring Boot's configuration file handling is quite flexible; and it often the case that properties may exist that are not bound to a `@ConfigurationProperties` bean. To support such cases, the annotation processor will automatically merge items from `META-INF/additional-spring-configuration-metadata.json` into the main meta-data file.

The format of the `additional-spring-configuration-metadata.json` file is exactly the same as the regular `spring-configuration-metadata.json`. The additional properties file is optional, if you don't have any additional properties, simply don't add it.

Appendix C. Auto-configuration classes

Here is a list of all auto configuration classes provided by Spring Boot with links to documentation and source code. Remember to also look at the autoconfig report in your application for more details of which features are switched on. (start the app with `--debug` or `-Ddebug`, or in an Actuator application use the `autoconfig` endpoint).

C.1 From the “spring-boot-autoconfigure” module

The following auto-configuration classes are from the `spring-boot-autoconfigure` module:

Configuration Class	Links
`ActiveMQAutoConfiguration`	javadoc
`AopAutoConfiguration`	javadoc
`BatchAutoConfiguration`	javadoc
`CloudAutoConfiguration`	javadoc
`DataSourceAutoConfiguration`	javadoc
`DataSourceTransactionManagerAutoConfiguration`	javadoc
`DeviceDelegatingViewResolverAutoConfiguration`	javadoc
`DeviceResolverAutoConfiguration`	javadoc
`DispatcherServletAutoConfiguration`	javadoc
`ElasticsearchAutoConfiguration`	javadoc
`ElasticsearchDataAutoConfiguration`	javadoc
`ElasticsearchRepositoriesAutoConfiguration`	javadoc
`EmbeddedServletContainerAutoConfiguration`	javadoc
`ErrorMvcAutoConfiguration`	javadoc
`FacebookAutoConfiguration`	javadoc
`FallbackWebSecurityAutoConfiguration`	javadoc
`FlywayAutoConfiguration`	javadoc
`FreeMarkerAutoConfiguration`	javadoc
`GroovyTemplateAutoConfiguration`	javadoc
`GsonAutoConfiguration`	javadoc
`HibernateJpaAutoConfiguration`	javadoc
`HornetQAutoConfiguration`	javadoc

Configuration Class	Links
HttpEncodingAutoConfiguration	javadoc
HttpMessageConvertersAutoConfiguration	javadoc
HypermediaAutoConfiguration	javadoc
IntegrationAutoConfiguration	javadoc
JacksonAutoConfiguration	javadoc
JerseyAutoConfiguration	javadoc
JmsAutoConfiguration	javadoc
JmxAutoConfiguration	javadoc
JndiConnectionFactoryAutoConfiguration	javadoc
JndiDataSourceAutoConfiguration	javadoc
JpaRepositoriesAutoConfiguration	javadoc
JtaAutoConfiguration	javadoc
LinkedInAutoConfiguration	javadoc
LiquibaseAutoConfiguration	javadoc
MailSenderAutoConfiguration	javadoc
MessageSourceAutoConfiguration	javadoc
MongoAutoConfiguration	javadoc
MongoDataAutoConfiguration	javadoc
MongoRepositoriesAutoConfiguration	javadoc
MultipartAutoConfiguration	javadoc
PersistenceExceptionTranslationAutoConfiguration	javadoc
PropertyPlaceholderAutoConfiguration	javadoc
RabbitAutoConfiguration	javadoc
ReactorAutoConfiguration	javadoc
RedisAutoConfiguration	javadoc
RepositoryRestMvcAutoConfiguration	javadoc
SecurityAutoConfiguration	javadoc
ServerPropertiesAutoConfiguration	javadoc
SitePreferenceAutoConfiguration	javadoc
SocialWebAutoConfiguration	javadoc

Configuration Class	Links
SolrAutoConfiguration	javadoc
SolrRepositoriesAutoConfiguration	javadoc
SpringDataWebAutoConfiguration	javadoc
ThymeleafAutoConfiguration	javadoc
TwitterAutoConfiguration	javadoc
VelocityAutoConfiguration	javadoc
WebMvcAutoConfiguration	javadoc
WebSocketAutoConfiguration	javadoc
XADataSourceAutoConfiguration	javadoc

C.2 From the “spring-boot-actuator” module

The following auto-configuration classes are from the `spring-boot-actuator` module:

Configuration Class	Links
AuditAutoConfiguration	javadoc
CrshAutoConfiguration	javadoc
EndpointAutoConfiguration	javadoc
EndpointMBeanExportAutoConfiguration	javadoc
EndpointWebMvcAutoConfiguration	javadoc
HealthIndicatorAutoConfiguration	javadoc
JolokiaAutoConfiguration	javadoc
ManagementSecurityAutoConfiguration	javadoc
ManagementServerPropertiesAutoConfiguration	javadoc
MetricFilterAutoConfiguration	javadoc
MetricRepositoryAutoConfiguration	javadoc
PublicMetricsAutoConfiguration	javadoc
TraceRepositoryAutoConfiguration	javadoc
TraceWebFilterAutoConfiguration	javadoc

Appendix D. The executable jar format

The `spring-boot-loader` modules allows Spring Boot to support executable jar and war files. If you're using the Maven or Gradle plugin, executable jars are automatically generated and you generally won't need to know the details of how they work.

If you need to create executable jars from a different build system, or if you are just curious about the underlying technology, this section provides some background.

D.1 Nested JARs

Java does not provide any standard way to load nested jar files (i.e. jar files that are themselves contained within a jar). This can be problematic if you are looking to distribute a self-contained application that you can just run from the command line without unpacking.

To solve this problem, many developers use "shaded" jars. A shaded jar simply packages all classes, from all jars, into a single 'uber jar'. The problem with shaded jars is that it becomes hard to see which libraries you are actually using in your application. It can also be problematic if the the same filename is used (but with different content) in multiple jars. Spring Boot takes a different approach and allows you to actually nest jars directly.

The executable jar file structure

Spring Boot Loader compatible jar files should be structured in the following way:

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-com
 |  +-mycompany
 |     + project
 |        +-YouClasses.class
 +-lib
    +-dependency1.jar
    +-dependency2.jar
```

Dependencies should be placed in a nested `lib` directory.

The executable war file structure

Spring Boot Loader compatible war files should be structured in the following way:

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
```

```
|        +-loader
|           +-<spring boot loader classes>
+-WEB-INF
   +-classes
   |  +-com
   |     +-mycompany
   |        +-project
   |           +-YouClasses.class
   +-lib
   |  +-dependency1.jar
   |  +-dependency2.jar
   +-lib-provided
      +-servlet-api.jar
      +-dependency3.jar
```

Dependencies should be placed in a nested `WEB-INF/lib` directory. Any dependencies that are required when running embedded but are not required when deploying to a traditional web container should be placed in `WEB-INF/lib-provided`.

D.2 Spring Boot's "JarFile" class

The core class used to support loading nested jars is `org.springframework.boot.loader.jar.JarFile`. It allows you load jar content from a standard jar file, or from nested child jar data. When first loaded, the location of each `JarEntry` is mapped to a physical file offset of the outer jar:

```
myapp.jar
+---------+---------------------+
|         | /lib/mylib.jar      |
| A.class |+---------+---------+|
|         || B.class | B.class ||
|         |+---------+---------+|
+---------+---------------------+
^          ^          ^
0063       3452       3980
```

The example above shows how `A.class` can be found in `myapp.jar` position `0063`. `B.class` from the nested jar can actually be found in `myapp.jar` position `3452` and `B.class` is at position `3980`.

Armed with this information, we can load specific nested entries by simply seeking to appropriate part if the outer jar. We don't need to unpack the archive and we don't need to read all entry data into memory.

Compatibility with the standard Java "JarFile"

Spring Boot Loader strives to remain compatible with existing code and libraries. `org.springframework.boot.loader.jar.JarFile` extends from `java.util.jar.JarFile` and should work as a drop-in replacement. The `RandomAccessJarFile.getURL()` method will return a `URL` that opens a `java.net.JarURLConnection` compatible connection. `RandomAccessJarFile` URLs can be used with Java's `URLClassLoader`.

D.3 Launching executable jars

The `org.springframework.boot.loader.Launcher` class is a special bootstrap class that is used as an executable jars main entry point. It is the actual `Main-Class` in your jar file and it's used to setup an appropriate `URLClassLoader` and ultimately call your `main()` method.

There are 3 launcher subclasses (`JarLauncher`, `WarLauncher` and `PropertiesLauncher`). Their purpose is to load resources (`.class` files etc.) from nested jar files or war files in directories (as opposed to explicitly on the classpath). In the case of the `[Jar|War]Launcher` the nested paths

are fixed (`lib/*.jar` and `lib-provided/*.jar` for the war case) so you just add extra jars in those locations if you want more. The `PropertiesLauncher` looks in `lib/` by default, but you can add additional locations by setting an environment variable `LOADER_PATH` or `loader.path` in `application.properties` (comma-separated list of directories or archives).

Launcher manifest

You need to specify an appropriate `Launcher` as the `Main-Class` attribute of `META-INF/MANIFEST.MF`. The actual class that you want to launch (i.e. the class that you wrote that contains a `main` method) should be specified in the `Start-Class` attribute.

For example, here is a typical `MANIFEST.MF` for an executable jar file:

```
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

For a war file, it would be:

```
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```

> **Note**
>
> You do not need to specify `Class-Path` entries in your manifest file, the classpath will be deduced from the nested jars.

Exploded archives

Certain PaaS implementations may choose to unpack archives before they run. For example, Cloud Foundry operates in this way. You can run an unpacked archive by simply starting the appropriate launcher:

```
$ unzip -q myapp.jar
$ java org.springframework.boot.loader.JarLauncher
```

D.4 PropertiesLauncher Features

`PropertiesLauncher` has a few special features that can be enabled with external properties (System properties, environment variables, manifest entries or `application.properties`).

Key	Purpose
`loader.path`	Comma-separated Classpath, e.g. `lib:${HOME}/app/lib`.
`loader.home`	Location of additional properties file, e.g. `/opt/app` (defaults to `${user.dir}`)
`loader.args`	Default arguments for the main method (space separated)
`loader.main`	Name of main class to launch, e.g. `com.app.Application`.
`loader.config.name`	Name of properties file, e.g. `loader` (defaults to `application`).
`loader.config.location`	Path to properties file, e.g. `classpath:loader.properties` (defaults to `application.properties`).

Key	Purpose
`loader.system`	Boolean flag to indicate that all properties should be added to System properties (defaults to `false`)

Manifest entry keys are formed by capitalizing initial letters of words and changing the separator to "-" from "." (e.g. `Loader-Path`). The exception is `loader.main` which is looked up as `Start-Class` in the manifest for compatibility with `JarLauncher`).

Environment variables can be capitalized with underscore separators instead of periods.

- `loader.home` is the directory location of an additional properties file (overriding the default) as long as `loader.config.location` is not specified.
- `loader.path` can contain directories (scanned recursively for jar and zip files), archive paths, or wildcard patterns (for the default JVM behavior).
- Placeholder replacement is done from System and environment variables plus the properties file itself on all values before use.

D.5 Executable jar restrictions

There are a number of restrictions that you need to consider when working with a Spring Boot Loader packaged application.

Zip entry compression

The `ZipEntry` for a nested jar must be saved using the `ZipEntry.STORED` method. This is required so that we can seek directly to individual content within the nested jar. The content of the nested jar file itself can still be compressed, as can any other entries in the outer jar.

System ClassLoader

Launched applications should use `Thread.getContextClassLoader()` when loading classes (most libraries and frameworks will do this by default). Trying to load nested jar classes via `ClassLoader.getSystemClassLoader()` will fail. Please be aware that `java.util.Logging` always uses the system classloader, for this reason you should consider a different logging implementation.

D.6 Alternative single jar solutions

If the above restrictions mean that you cannot use Spring Boot Loader the following alternatives could be considered:

- Maven Shade Plugin
- JarClassLoader
- OneJar

Appendix E. Dependency versions

The table below provides details of all of the dependency versions that are provided by Spring Boot in its CLI, Maven dependency management and Gradle plugin. When you declare a dependency on one of these artifacts without declaring a version the version that is listed in the table will be used.

Group ID	Artifact ID	Version
ch.qos.logback	logback-classic	1.1.2
com.atomikos	transactions-jdbc	3.9.3
com.atomikos	transactions-jms	3.9.3
com.atomikos	transactions-jta	3.9.3
com.fasterxml.jackson.core	jackson-annotations	2.4.4
com.fasterxml.jackson.core	jackson-core	2.4.4
com.fasterxml.jackson.core	jackson-databind	2.4.4
com.fasterxml.jackson.dataformat	jackson-dataformat-xml	2.4.4
com.fasterxml.jackson.datatype	jackson-datatype-jdk8	2.4.4
com.fasterxml.jackson.datatype	jackson-datatype-joda	2.4.4
com.fasterxml.jackson.datatype	jackson-datatype-jsr310	2.4.4
com.gemstone.gemfire	gemfire	7.0.2
com.github.mxab.thymeleaf.extras	thymeleaf-extras-data-attribute	1.3
com.google.code.gson	gson	2.3
com.googlecode.json-simple	json-simple	1.1.1
com.h2database	h2	1.4.182
com.jayway.jsonpath	json-path	0.9.1
com.sun.mail	javax.mail	1.5.2
com.zaxxer	HikariCP	2.2.5
com.zaxxer	HikariCP-java6	2.2.5
commons-beanutils	commons-beanutils	1.9.2
commons-collections	commons-collections	3.2.1
commons-dbcp	commons-dbcp	1.4
commons-digester	commons-digester	2.1
commons-pool	commons-pool	1.6

Group ID	Artifact ID	Version
io.dropwizard.metrics	metrics-core	3.1.0
io.dropwizard.metrics	metrics-ganglia	3.1.0
io.dropwizard.metrics	metrics-graphite	3.1.0
io.dropwizard.metrics	metrics-servlets	3.1.0
io.undertow	undertow-core	1.1.1.Final
io.undertow	undertow-servlet	1.1.1.Final
io.undertow	undertow-websockets-jsr	1.1.1.Final
javax.cache	cache-api	1.0.0
javax.jms	jms-api	1.1-rev-1
javax.mail	javax.mail-api	1.5.2
javax.servlet	javax.servlet-api	3.1.0
javax.servlet	jstl	1.2
jaxen	jaxen	1.1.6
joda-time	joda-time	2.5
junit	junit	4.12
log4j	log4j	1.2.17
mysql	mysql-connector-java	5.1.34
nz.net.ultraq.thymeleaf	thymeleaf-layout-dialect	1.2.7
org.apache.activemq	activemq-broker	5.10.0
org.apache.activemq	activemq-client	5.10.0
org.apache.activemq	activemq-jms-pool	5.10.0
org.apache.activemq	activemq-pool	5.10.0
org.apache.commons	commons-dbcp2	2.0.1
org.apache.commons	commons-pool2	2.2
org.apache.httpcomponents	httpasyncclient	4.0.2
org.apache.httpcomponents	httpclient	4.3.6
org.apache.httpcomponents	httpmime	4.3.6
org.apache.logging.log4j	log4j-api	2.1
org.apache.logging.log4j	log4j-core	2.1
org.apache.logging.log4j	log4j-slf4j-impl	2.1

Group ID	Artifact ID	Version
org.apache.solr	solr-solrj	4.7.2
org.apache.tomcat	tomcat-jdbc	8.0.15
org.apache.tomcat	tomcat-jsp-api	8.0.15
org.apache.tomcat.embed	tomcat-embed-core	8.0.15
org.apache.tomcat.embed	tomcat-embed-el	8.0.15
org.apache.tomcat.embed	tomcat-embed-jasper	8.0.15
org.apache.tomcat.embed	tomcat-embed-logging-juli	8.0.15
org.apache.tomcat.embed	tomcat-embed-websocket	8.0.15
org.apache.velocity	velocity	1.7
org.apache.velocity	velocity-tools	2.0
org.aspectj	aspectjrt	1.8.4
org.aspectj	aspectjtools	1.8.4
org.aspectj	aspectjweaver	1.8.4
org.codehaus.btm	btm	2.1.4
org.codehaus.groovy	groovy	2.3.8
org.codehaus.groovy	groovy-all	2.3.8
org.codehaus.groovy	groovy-ant	2.3.8
org.codehaus.groovy	groovy-bsf	2.3.8
org.codehaus.groovy	groovy-console	2.3.8
org.codehaus.groovy	groovy-docgenerator	2.3.8
org.codehaus.groovy	groovy-groovydoc	2.3.8
org.codehaus.groovy	groovy-groovysh	2.3.8
org.codehaus.groovy	groovy-jmx	2.3.8
org.codehaus.groovy	groovy-json	2.3.8
org.codehaus.groovy	groovy-jsr223	2.3.8
org.codehaus.groovy	groovy-nio	2.3.8
org.codehaus.groovy	groovy-servlet	2.3.8
org.codehaus.groovy	groovy-sql	2.3.8
org.codehaus.groovy	groovy-swing	2.3.8
org.codehaus.groovy	groovy-templates	2.3.8

Group ID	Artifact ID	Version
org.codehaus.groovy	groovy-test	2.3.8
org.codehaus.groovy	groovy-testng	2.3.8
org.codehaus.groovy	groovy-xml	2.3.8
org.codehaus.janino	janino	2.6.1
org.crashub	crash.cli	1.3.0
org.crashub	crash.connectors.ssh	1.3.0
org.crashub	crash.connectors.telnet	1.3.0
org.crashub	crash.embed.spring	1.3.0
org.crashub	crash.plugins.cron	1.3.0
org.crashub	crash.plugins.mail	1.3.0
org.crashub	crash.shell	1.3.0
org.eclipse.jetty	jetty-annotations	9.2.4.v20141103
org.eclipse.jetty	jetty-jsp	9.2.4.v20141103
org.eclipse.jetty	jetty-util	9.2.4.v20141103
org.eclipse.jetty	jetty-webapp	9.2.4.v20141103
org.eclipse.jetty.orbit	javax.servlet.jsp	2.2.0.v201112011158
org.eclipse.jetty.websocket	javax-websocket-server-impl	9.2.4.v20141103
org.eclipse.jetty.websocket	websocket-server	9.2.4.v20141103
org.flywaydb	flyway-core	3.0
org.freemarker	freemarker	2.3.21
org.glassfish	javax.el	3.0.0
org.glassfish.jersey.containers	jersey-container-servlet	2.13
org.glassfish.jersey.containers	jersey-container-servlet-core	2.13
org.glassfish.jersey.core	jersey-server	2.13
org.glassfish.jersey.ext	jersey-spring3	2.13
org.glassfish.jersey.media	jersey-media-json-jackson	2.13
org.hamcrest	hamcrest-core	1.3
org.hamcrest	hamcrest-library	1.3

Group ID	Artifact ID	Version
org.hibernate	hibernate-ehcache	4.3.7.Final
org.hibernate	hibernate-entitymanager	4.3.7.Final
org.hibernate	hibernate-envers	4.3.7.Final
org.hibernate	hibernate-jpamodelgen	4.3.7.Final
org.hibernate	hibernate-validator	5.1.3.Final
org.hornetq	hornetq-jms-client	2.4.5.Final
org.hornetq	hornetq-jms-server	2.4.5.Final
org.hsqldb	hsqldb	2.3.2
org.javassist	javassist	3.18.1-GA
org.jdom	jdom2	2.0.5
org.jolokia	jolokia-core	1.2.3
org.liquibase	liquibase-core	3.3.0
org.mockito	mockito-core	1.10.8
org.mongodb	mongo-java-driver	2.12.4
org.projectreactor	reactor-core	1.1.5.RELEASE
org.projectreactor	reactor-groovy	1.1.5.RELEASE
org.projectreactor	reactor-groovy-extensions	1.1.5.RELEASE
org.projectreactor	reactor-logback	1.1.5.RELEASE
org.projectreactor	reactor-net	1.1.5.RELEASE
org.projectreactor.spring	reactor-spring-context	1.1.3.RELEASE
org.projectreactor.spring	reactor-spring-core	1.1.3.RELEASE
org.projectreactor.spring	reactor-spring-messaging	1.1.3.RELEASE
org.projectreactor.spring	reactor-spring-webmvc	1.1.3.RELEASE
org.slf4j	jcl-over-slf4j	1.7.7
org.slf4j	jul-to-slf4j	1.7.7
org.slf4j	log4j-over-slf4j	1.7.7
org.slf4j	slf4j-api	1.7.7
org.slf4j	slf4j-jdk14	1.7.7
org.slf4j	slf4j-log4j12	1.7.7

Group ID	Artifact ID	Version
org.spockframework	spock-core	0.7-groovy-2.0
org.spockframework	spock-spring	0.7-groovy-2.0
org.springframework	spring-aop	4.1.3.RELEASE
org.springframework	spring-aspects	4.1.3.RELEASE
org.springframework	spring-beans	4.1.3.RELEASE
org.springframework	spring-context	4.1.3.RELEASE
org.springframework	spring-context-support	4.1.3.RELEASE
org.springframework	spring-core	4.1.3.RELEASE
org.springframework	spring-expression	4.1.3.RELEASE
org.springframework	spring-instrument	4.1.3.RELEASE
org.springframework	spring-instrument-tomcat	4.1.3.RELEASE
org.springframework	spring-jdbc	4.1.3.RELEASE
org.springframework	spring-jms	4.1.3.RELEASE
org.springframework	springloaded	1.2.1.RELEASE
org.springframework	spring-messaging	4.1.3.RELEASE
org.springframework	spring-orm	4.1.3.RELEASE
org.springframework	spring-oxm	4.1.3.RELEASE
org.springframework	spring-test	4.1.3.RELEASE
org.springframework	spring-tx	4.1.3.RELEASE
org.springframework	spring-web	4.1.3.RELEASE
org.springframework	spring-webmvc	4.1.3.RELEASE
org.springframework	spring-webmvc-portlet	4.1.3.RELEASE
org.springframework	spring-websocket	4.1.3.RELEASE
org.springframework.amqp	spring-amqp	1.4.0.RELEASE
org.springframework.amqp	spring-erlang	1.4.0.RELEASE
org.springframework.amqp	spring-rabbit	1.4.0.RELEASE
org.springframework.batch	spring-batch-core	3.0.2.RELEASE
org.springframework.batch	spring-batch-infrastructure	3.0.2.RELEASE

Group ID	Artifact ID	Version
org.springframework.batch	spring-batch-integration	3.0.2.RELEASE
org.springframework.batch	spring-batch-test	3.0.2.RELEASE
org.springframework.boot	spring-boot	1.2.0.RELEASE
org.springframework.boot	spring-boot	1.2.0.RELEASE
org.springframework.boot	spring-boot-actuator	1.2.0.RELEASE
org.springframework.boot	spring-boot-autoconfigure	1.2.0.RELEASE
org.springframework.boot	spring-boot-configuration-processor	1.2.0.RELEASE
org.springframework.boot	spring-boot-dependency-tools	1.2.0.RELEASE
org.springframework.boot	spring-boot-loader	1.2.0.RELEASE
org.springframework.boot	spring-boot-loader-tools	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-actuator	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-amqp	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-aop	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-batch	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-cloud-connectors	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-data-elasticsearch	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-data-gemfire	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-data-jpa	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-data-mongodb	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-data-rest	1.2.0.RELEASE

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-data-solr	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-freemarker	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-groovy-templates	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-hornetq	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-integration	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-jdbc	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-jersey	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-jetty	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-jta-atomikos	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-jta-bitronix	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-log4j	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-log4j2	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-logging	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-mail	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-mobile	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-redis	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-remote-shell	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-security	1.2.0.RELEASE

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-social-facebook	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-social-linkedin	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-social-twitter	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-test	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-thymeleaf	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-tomcat	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-undertow	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-velocity	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-web	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-websocket	1.2.0.RELEASE
org.springframework.boot	spring-boot-starter-ws	1.2.0.RELEASE
org.springframework.cloud	spring-cloud-cloudfoundry-connector	1.1.0.RELEASE
org.springframework.cloud	spring-cloud-core	1.1.0.RELEASE
org.springframework.cloud	spring-cloud-heroku-connector	1.1.0.RELEASE
org.springframework.cloud	spring-cloud-localconfig-connector	1.1.0.RELEASE
org.springframework.cloud	spring-cloud-spring-service-connector	1.1.0.RELEASE
org.springframework.data	spring-cql	1.1.1.RELEASE
org.springframework.data	spring-data-cassandra	1.1.1.RELEASE
org.springframework.data	spring-data-commons	1.9.1.RELEASE
org.springframework.data	spring-data-couchbase	1.2.1.RELEASE
org.springframework.data	spring-data-elasticsearch	1.1.1.RELEASE

Group ID	Artifact ID	Version
org.springframework.data	spring-data-gemfire	1.5.1.RELEASE
org.springframework.data	spring-data-jpa	1.7.1.RELEASE
org.springframework.data	spring-data-mongodb	1.6.1.RELEASE
org.springframework.data	spring-data-mongodb-cross-store	1.6.1.RELEASE
org.springframework.data	spring-data-mongodb-log4j	1.6.1.RELEASE
org.springframework.data	spring-data-neo4j	3.2.1.RELEASE
org.springframework.data	spring-data-redis	1.4.1.RELEASE
org.springframework.data	spring-data-rest-core	2.2.1.RELEASE
org.springframework.data	spring-data-rest-webmvc	2.2.1.RELEASE
org.springframework.data	spring-data-solr	1.3.1.RELEASE
org.springframework.hateoas	spring-hateoas	0.16.0.RELEASE
org.springframework.integration	spring-integration-amqp	4.1.0.RELEASE
org.springframework.integration	spring-integration-core	4.1.0.RELEASE
org.springframework.integration	spring-integration-event	4.1.0.RELEASE
org.springframework.integration	spring-integration-feed	4.1.0.RELEASE
org.springframework.integration	spring-integration-file	4.1.0.RELEASE
org.springframework.integration	spring-integration-ftp	4.1.0.RELEASE
org.springframework.integration	spring-integration-gemfire	4.1.0.RELEASE
org.springframework.integration	spring-integration-groovy	4.1.0.RELEASE
org.springframework.integration	spring-integration-http	4.1.0.RELEASE
org.springframework.integration	spring-integration-ip	4.1.0.RELEASE
org.springframework.integration	spring-integration-jdbc	4.1.0.RELEASE
org.springframework.integration	spring-integration-jms	4.1.0.RELEASE
org.springframework.integration	spring-integration-jmx	4.1.0.RELEASE
org.springframework.integration	spring-integration-jpa	4.1.0.RELEASE
org.springframework.integration	spring-integration-mail	4.1.0.RELEASE

Group ID	Artifact ID	Version
org.springframework.integration	spring-integration-mongodb	4.1.0.RELEASE
org.springframework.integration	spring-integration-mqtt	4.1.0.RELEASE
org.springframework.integration	spring-integration-redis	4.1.0.RELEASE
org.springframework.integration	spring-integration-rmi	4.1.0.RELEASE
org.springframework.integration	spring-integration-scripting	4.1.0.RELEASE
org.springframework.integration	spring-integration-security	4.1.0.RELEASE
org.springframework.integration	spring-integration-sftp	4.1.0.RELEASE
org.springframework.integration	spring-integration-stream	4.1.0.RELEASE
org.springframework.integration	spring-integration-syslog	4.1.0.RELEASE
org.springframework.integration	spring-integration-test	4.1.0.RELEASE
org.springframework.integration	spring-integration-twitter	4.1.0.RELEASE
org.springframework.integration	spring-integration-websocket	4.1.0.RELEASE
org.springframework.integration	spring-integration-ws	4.1.0.RELEASE
org.springframework.integration	spring-integration-xml	4.1.0.RELEASE
org.springframework.integration	spring-integration-xmpp	4.1.0.RELEASE
org.springframework.mobile	spring-mobile-device	1.1.3.RELEASE
org.springframework.security	spring-security-acl	3.2.5.RELEASE
org.springframework.security	spring-security-aspects	3.2.5.RELEASE
org.springframework.security	spring-security-cas	3.2.5.RELEASE
org.springframework.security	spring-security-config	3.2.5.RELEASE
org.springframework.security	spring-security-core	3.2.5.RELEASE
org.springframework.security	spring-security-crypto	3.2.5.RELEASE
org.springframework.security	spring-security-jwt	1.0.2.RELEASE
org.springframework.security	spring-security-ldap	3.2.5.RELEASE
org.springframework.security	spring-security-openid	3.2.5.RELEASE

Group ID	Artifact ID	Version
org.springframework.security	spring-security-remoting	3.2.5.RELEASE
org.springframework.security	spring-security-taglibs	3.2.5.RELEASE
org.springframework.security	spring-security-web	3.2.5.RELEASE
org.springframework.social	spring-social-config	1.1.0.RELEASE
org.springframework.social	spring-social-core	1.1.0.RELEASE
org.springframework.social	spring-social-facebook	1.1.1.RELEASE
org.springframework.social	spring-social-facebook-web	1.1.1.RELEASE
org.springframework.social	spring-social-linkedin	1.0.1.RELEASE
org.springframework.social	spring-social-security	1.1.0.RELEASE
org.springframework.social	spring-social-twitter	1.1.0.RELEASE
org.springframework.social	spring-social-web	1.1.0.RELEASE
org.springframework.ws	spring-ws-core	2.2.0.RELEASE
org.springframework.ws	spring-ws-security	2.2.0.RELEASE
org.springframework.ws	spring-ws-support	2.2.0.RELEASE
org.springframework.ws	spring-ws-test	2.2.0.RELEASE
org.thymeleaf	thymeleaf	2.1.3.RELEASE
org.thymeleaf	thymeleaf-spring4	2.1.3.RELEASE
org.thymeleaf.extras	thymeleaf-extras-springsecurity3	2.1.1.RELEASE
org.yaml	snakeyaml	1.14
redis.clients	jedis	2.5.2
wsdl4j	wsdl4j	1.6.3