Spring Boot Reference Guide

1.4.0.M2

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

Table of Contents

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.

1. About the documentation

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.

2. Getting help

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 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.

3. First steps

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

4. Working with Spring Boot

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

5. Learning about Spring Boot features

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

6. Moving to production

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

7. Advanced topics

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

- **Deploy Spring Boot Applications:** Cloud Deployment | OS Service
- **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.

8. Introducing Spring Boot

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.

9. System Requirements

By default, Spring Boot 1.4.0.M2 requires Java 7 and Spring Framework 4.3.0.RC1 or above. You can use Spring Boot with Java 6 with some additional configuration. See Section 79.11, “How to use Java 6” for more details. Explicit build support is provided for Maven (3.2+) and Gradle (1.12+).

Tip

Although you can use Spring Boot with Java 6 or 7, 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.

10. Installing Spring Boot

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>1.4.0.M2</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>

    <!-- Add Spring repositories -->
    <!-- (you don't need this if you are using a .RELEASE version) -->
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </repository>
    </repositories>
    <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>
</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:1.4.0.M2")
    }
}

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-1.4.0.M2-bin.zip
- spring-boot-cli-1.4.0.M2-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 SDKMAN!

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

```
$ sdk install springboot
$ spring --version
Spring Boot v1.4.0.M2
```

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

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-cli-1.4.0.M2-bin/
spring-1.4.0.M2/
$ sdk default springboot dev
$ spring --version
Spring CLI v1.4.0.M2
```

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

You can see it by doing this:

```
$ sdk ls springboot

================================================================================
Available Springboot Versions
================================================================================
> + dev
* 1.4.0.M2

================================================================================
+ - 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.

MacPorts installation

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

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

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 SDKMAN!

```
$ . ~/.sdkman/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 or MacPorts, 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.

11. Developing your first Spring Boot application

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.

You can shortcut the steps below by going to start.spring.io and choosing the `web` starter from the dependencies searcher. This will automatically generate a new project structure so that you can start coding right the way. Check the documentation for more details.

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>1.4.0.M2</version>
    </parent>

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

    <!-- (you don't need this if you are using a .RELEASE version) -->
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/snapshot</url>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <url>http://repo.spring.io/milestone</url>
        </repository>
    </repositories>
    <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>
</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`:

```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 ::  (v1.4.0.M2)
....... . . .
....... . . . (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 “uber” jars. An uber jar simply packages all classes, from all jars, into a single archive. The problem with this approach is that it becomes hard to see which libraries you are actually using in your application. It can also be problematic if the same filename is used (but with different content) in multiple jars.

Spring Boot takes a different approach and 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:1.4.0.M2: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 ::  (v1.4.0.M2)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.536 seconds (JVM running for 2.864)
```

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

12. What to read next

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.

The Spring Boot repository has also a bunch of samples you can run. The samples are independent of the rest of the code (that is you don't need to build the rest to run or use the samples).

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 how to run your applications. We also cover some Spring Boot best practices. Although there is nothing particularly special about Spring Boot (it is just another library that you can consume), there are a few recommendations that, when followed, 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.

13. Build systems

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 Dependency management

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

Note

You can still specify a version and override Spring Boot’s recommendations if you feel that’s necessary.

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

Warning

Each release of Spring Boot is associated with a base version of the Spring Framework so we **highly** recommend you to not specify its version on your own.

13.2 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` including profile-specific files (e.g. `application-foo.properties` and `application-foo.yml`)

On the last point: since the default config 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>1.4.0.M2</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.

With that setup, you can also override individual dependencies by overriding a property in your own project. For instance, to upgrade to another Spring Data release train you'd add the following to your `pom.xml`.

```
<properties>
    <spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version>
</properties>
```

Tip

Check the [`spring-boot-dependencies` pom]() for a list of supported properties.

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>1.4.0.M2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

That setup does not allow you to override individual dependencies using a property as explained above. To achieve the same result, you'd need to add an entry in the `dependencyManagement` of your project **before** the `spring-boot-dependencies` entry. For instance, to upgrade to another Spring Data release train you'd add the following to your `pom.xml`.

```
<dependencyManagement>
    <dependencies>
        <!-- Override Spring Data release train provided by Spring Boot -->
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-releasetrain</artifactId>
            <version>Fowler-SR2</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>1.4.0.M2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Note

In the example above, we specify a *BOM* but any dependency type can be overridden that way.

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.

13.3 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 {
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.4.0.M2")
}
```

The `spring-boot-gradle-plugin` is also available and provides tasks to create executable jars and run projects from source. It also provides dependency management that, among other capabilities, allows you to omit the version number for any dependencies that are managed by Spring Boot:

```
buildscript {
    repositories {
        maven { url "http://repo.spring.io/snapshot" }
        maven { url "http://repo.spring.io/milestone" }
    }

    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.4.0.M2")
    }
}

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

repositories {
    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")
}
```

13.4 Ant

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

To declare dependencies a typical `ivy.xml` file will look something like this:

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

A typical `build.xml` will look like this:

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

    <property name="spring-boot.version" value="1.3.0.BUILD-SNAPSHOT" />

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

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

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

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

Tip

See the *Section 79.10, “Build an executable archive from Ant without using spring-boot-antlib”* “How-to” if you don’t want to use the `spring-boot-antlib` module.

13.5 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 **official** starters follow a similar naming pattern; `spring-boot-starter-*`, where `*` is a particular type of application. This naming structure is intended to help when you need to find a starter. The Maven integration in many IDEs 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.

As explained in the Creating your own starter section, third party starters should not start with `spring-boot` as it is reserved for official Spring Boot artifacts. A third-party starter for `acme` will be typically named `acme-spring-boot-starter`.

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

Table 13.1. Spring Boot application starters

Name	Description	Pom
`spring-boot-starter-test`	Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito	Pom

Name	Description	Pom
`spring-boot-starter-mobile`	Starter for building web applications using Spring Mobile	Pom
`spring-boot-starter-social-twitter`	Starter for using Spring Social Twitter	Pom
`spring-boot-starter-cache`	Starter for using Spring Framework's caching support	Pom
`spring-boot-starter-jta-atomikos`	Starter for JTA transactions using Atomikos	Pom
`spring-boot-starter-aop`	Starter for aspect-oriented programming with Spring AOP and AspectJ	Pom
`spring-boot-starter-web`	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container	Pom
`spring-boot-starter-data-elasticsearch`	Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch	Pom
`spring-boot-starter-jdbc`	Starter for using JDBC with the Tomcat JDBC connection pool	Pom
`spring-boot-starter-batch`	Starter for using Spring Batch, including HSQLDB in-memory database	Pom
`spring-boot-starter-social-facebook`	Starter for using Spring Social Facebook	Pom
`spring-boot-starter-jta-narayana`	Spring Boot Narayana JTA Starter	Pom
`spring-boot-starter-thymeleaf`	Starter for building MVC web applications using Thymeleaf views	Pom
`spring-boot-starter-mail`	Starter for using Java Mail and Spring Framework's email sending support	Pom
`spring-boot-starter-jta-bitronix`	Starter for JTA transactions using Bitronix	Pom

Name	Description	Pom
`spring-boot-starter-data-mongodb`	Starter for using MongoDB document-oriented database and Spring Data MongoDB	Pom
`spring-boot-starter-validation`	Starter for using Java Bean Validation with Hibernate Validator	Pom
`spring-boot-starter-jooq`	Starter for using jOOQ to access SQL databases. An alternative to `spring-boot-starter-data-jpa` or `spring-boot-starter-jdbc`	Pom
`spring-boot-starter-data-cassandra`	Starter for using Cassandra distributed database and Spring Data Cassandra	Pom
`spring-boot-starter-hateoas`	Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS	Pom
`spring-boot-starter-integration`	Starter for using Spring Integration	Pom
`spring-boot-starter-data-solr`	Starter for using the Apache Solr search platform with Spring Data Solr	Pom
`spring-boot-starter-freemarker`	Starter for building MVC web applications using Freemarker views	Pom
`spring-boot-starter-jersey`	Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to `spring-boot-starter-web`	Pom
`spring-boot-starter`	Core starter, including auto-configuration support, logging and YAML	Pom
`spring-boot-starter-data-couchbase`	Starter for using Couchbase document-oriented database and Spring Data Couchbase	Pom
`spring-boot-starter-artemis`	Starter for JMS messaging using Apache Artemis	Pom
`spring-boot-starter-cloud-connectors`	Starter for using Spring Cloud Connectors which simplifies	Pom

Name	Description	Pom
	connecting to services in cloud platforms like Cloud Foundry and Heroku	
`spring-boot-starter-social-linkedin`	Stater for using Spring Social LinkedIn	Pom
`spring-boot-starter-velocity`	Starter for building MVC web applications using Velocity views. Deprecated since 1.4	Pom
`spring-boot-starter-data-rest`	Starter for exposing Spring Data repositories over REST using Spring Data REST	Pom
`spring-boot-starter-data-gemfire`	Starter for using GemFire distributed data store and Spring Data GemFire	Pom
`spring-boot-starter-groovy-templates`	Starter for building MVC web applications using Groovy Templates views	Pom
`spring-boot-starter-amqp`	Starter for using Spring AMQP and Rabbit MQ	Pom
`spring-boot-starter-hornetq`	Starter for JMS messaging using HornetQ	Pom
`spring-boot-starter-ws`	Starter for using Spring Web Services	Pom
`spring-boot-starter-security`	Starter for using Spring Security	Pom
`spring-boot-starter-data-redis`	Starter for using Redis key-value data store with Spring Data Redis and the Jedis client	Pom
`spring-boot-starter-websocket`	Starter for building WebSocket applications using Spring Framework’s WebSocket support	Pom
`spring-boot-starter-mustache`	Starter for building MVC web applications using Mustache views	Pom
`spring-boot-starter-data-neo4j`	Starter for using Neo4j graph database and Spring Data Neo4j	Pom
`spring-boot-starter-data-jpa`	Starter for using Spring Data JPA with Hibernate	Pom

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

Table 13.2. Spring Boot production starters

Name	**Description**	**Pom**
`spring-boot-starter-actuator`	Starter for using Spring Boot’s Actuator which provides production ready features to help you monitor and manage your application	Pom
`spring-boot-starter-remote-shell`	Starter for using the CRaSH remote shell to monitor and manage your application over SSH	Pom

Finally, Spring Boot also 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**	**Pom**
`spring-boot-starter-undertow`	Starter for using Undertow as the embedded servlet container. An alternative to `spring-boot-starter-tomcat`	Pom
`spring-boot-starter-logging`	Starter for logging using Logback. Default logging starter	Pom
`spring-boot-starter-tomcat`	Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by `spring-boot-starter-web`	Pom
`spring-boot-starter-jetty`	Starter for using Jetty as the embedded servlet container. An alternative to `spring-boot-starter-tomcat`	Pom
`spring-boot-starter-log4j2`	Starter for using Log4j2 for logging. An alternative to `spring-boot-starter-logging`	Pom

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

14. Structuring your code

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`.

```java
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);
    }

}
```

15. Configuration classes

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-based 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 pick up 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.

16. Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, If `HSQLDB` is on your classpath, and you have not manually configured any database connection beans, then 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, start your application with the `--debug` switch. This will enables debug logs for a selection of core loggers and 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.

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

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

If the class is not on the classpath, you can use the `excludeName` attribute of the annotation and specify the fully qualified name instead. Finally, you can also control the list of auto-configuration classes to exclude via the `spring.autoconfigure.exclude` property.

> **Tip**
>
> You can define exclusions both at the annotation level and using the property.

17. Spring Beans and dependency injection

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. 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.

18. Using the @SpringBootApplication annotation

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);
    }

}
```

Note

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

19. Running your application

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 just like in your IDE.

```
$ 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 `bootRun` task which can be used to run your application in an exploded form. The `bootRun` task is added whenever you import the `spring-boot-gradle-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 JRebel or the Spring Loaded project can be used. The `spring-boot-devtools` module also includes support for quick application restarts.

See the Chapter 20, *Developer tools* section below and the Hot swapping “How-to” for details.

20. Developer tools

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

Maven.

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

Gradle.

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

Note

Developer tools are automatically disabled when running a fully packaged application. If your application is launched using `java -jar` or if it's started using a special classloader, then it is considered a "production application". Flagging the dependency as optional is a best practice that prevents devtools from being transitively applied to other modules using your project. Gradle does not support `optional` dependencies out-of-the-box so you may want to have a look to the `propdeps-plugin` in the meantime.

Tip

If you want to ensure that devtools is never included in a production build, you can use the `excludeDevtools` build property to completely remove the JAR. The property is supported with both the Maven and Gradle plugins.

20.1 Property defaults

Several of the libraries supported by Spring Boot use caches to improve performance. For example, Thymeleaf will cache templates to save repeatedly parsing XML source files. Whilst caching is very beneficial in production, it can be counter productive during development. If you make a change to a template file in your IDE, you'll likely want to immediately see the result.

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

Tip

For a complete list of the properties that are applied see DevToolsPropertyDefaultsPostProcessor.

20.2 Automatic restart

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

Triggering a restart

As DevTools monitors classpath resources, the only way to trigger a restart is to update the classpath. The way in which you cause the classpath to be updated depends on the IDE that you are using. In Eclipse, saving a modified file will cause the classpath to be updated and trigger a restart. In IntelliJ IDEA, building the project (`Build # Make Project`) will have the same effect.

Note

You can also start your application via the supported build plugins (i.e. Maven and Gradle) as long as forking is enabled since DevTools need an isolated application classloader to operate properly. You can force the plugin to fork the process as follows:

Maven.

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

Gradle.

```
bootRun {
    addResources = true
}
```

Tip

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

Note

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

Note

When deciding if an entry on the classpath should trigger a restart when it changes, DevTools automatically ignores projects named `spring-boot`, `spring-boot-devtools`, `spring-boot-autoconfigure`, `spring-boot-actuator`, and `spring-boot-starter`.

Restart vs Reload

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

If you find that restarts aren't quick enough for your applications, or you encounter classloading issues, you could consider reloading technologies such as JRebel from ZeroTurnaround. These work by rewriting classes as they are loaded to make them more amenable to reloading. Spring Loaded provides another option, however it doesn't support as many frameworks and it isn't commercially supported.

Excluding resources

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

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

Tip

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

Watching additional paths

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

Disabling restart

If you don't want to use the restart feature you can disable it using the `spring.devtools.restart.enabled` property. In most cases you can set this in your `application.properties` (this will still initialize the restart classloader but it won't watch for file changes).

If you need to *completely* disable restart support, for example, because it doesn't work with a specific library, you need to set a `System` property before calling `SpringApplication.run(…)`. For example:

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

Using a trigger file

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

To use a trigger file use the `spring.devtools.restart.trigger-file` property.

Tip

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

Customizing the restart classloader

As described in the Restart vs Reload section above, restart functionality is implemented by using two classloaders. For most applications this approach works well, however, sometimes it can cause classloading issues.

By default, any open project in your IDE will be loaded using the "restart" classloader, and any regular `.jar` file will be loaded using the "base" classloader. If you work on a multi-module project, and not each module is imported into your IDE, you may need to customize things. To do this you can create a `META-INF/spring-devtools.properties` file.

The `spring-devtools.properties` file can contain `restart.exclude.` and `restart.include.` prefixed properties. The `include` elements are items that should be pulled up into the "restart" classloader, and the `exclude` elements are items that should be pushed down into the "base" classloader. The value of the property is a regex pattern that will be applied to the classpath.

For example:

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

Note

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

Tip

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

Known limitations

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

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

20.3 LiveReload

The `spring-boot-devtools` module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. LiveReload browser extensions are freely available for Chrome, Firefox and Safari from livereload.com.

If you don't want to start the LiveReload server when your application runs you can set the `spring.devtools.livereload.enabled` property to `false`.

Note

You can only run one LiveReload server at a time, if you start multiple applications from your IDE only the first will have livereload support.

20.4 Global settings

You can configure global devtools settings by adding a file named `.spring-boot-devtools.properties` to your `$HOME` folder (note that the filename starts with "."). Any properties added to this file will apply to *all* Spring Boot applications on your machine that use devtools. For example, to configure restart to always use a trigger file, you would add the following:

~/.spring-boot-devtools.properties.

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

20.5 Remote applications

The Spring Boot developer tools are not just limited to local development. You can also use several features when running applications remotely. Remote support is opt-in, to enable it you need to set a `spring.devtools.remote.secret` property. For example:

```
spring.devtools.remote.secret=mysecret
```

Warning

Enabling `spring-boot-devtools` on a remote application is a security risk. You should never enable support on a production deployment.

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

Running the remote client application

The remote client application is designed to be run from within you IDE. You need to run `org.springframework.boot.devtools.RemoteSpringApplication` using the same classpath as the remote project that you're connecting to. The *non-option* argument passed to the application should be the remote URL that you are connecting to.

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

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

A running remote client will look like this:

```
  .   ____          _                                              __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _          ___               _      \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` |        | _ \___ _ __  ___| |_ ___ \ \ \ \
 \\/  ___)| |_)| | | | | || (_| []::::::[]   / -_) '  \/ _ \  _/ -_) ) ) ) )
  '  |____| .__|_| |_|_| |_\__, |        |_|_\___|_|_|_\___/\__\___|/ / / /
 =========|_|==============|___/===================================/_/_/_/
 :: Spring Boot Remote :: 1.4.0.M2

2015-06-10 18:25:06.632  INFO 14938 --- [           main] o.s.b.devtools.RemoteSpringApplication   :
 Starting RemoteSpringApplication on pwmbp with PID 14938 (/Users/pwebb/projects/spring-boot/code/
spring-boot-devtools/target/classes started by pwebb in /Users/pwebb/projects/spring-boot/code/spring-
boot-samples/spring-boot-sample-devtools)
2015-06-10 18:25:06.671  INFO 14938 --- [           main] s.c.a.AnnotationConfigApplicationContext :
 Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@2a17b7b6: startup
 date [Wed Jun 10 18:25:06 PDT 2015]; root of context hierarchy
2015-06-10 18:25:07.043  WARN 14938 --- [           main] o.s.b.d.r.c.RemoteClientConfiguration    : The
 connection to http://localhost:8080 is insecure. You should use a URL starting with 'https://'.
2015-06-10 18:25:07.074  INFO 14938 --- [           main] o.s.b.d.a.OptionalLiveReloadServer       :
 LiveReload server is running on port 35729
2015-06-10 18:25:07.130  INFO 14938 --- [           main] o.s.b.devtools.RemoteSpringApplication   :
 Started RemoteSpringApplication in 0.74 seconds (JVM running for 1.105)
```

Note

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

Tip

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

Tip

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

Remote update

The remote client will monitor your application classpath for changes in the same way as the local restart. Any updated resource will be pushed to the remote application and *(if required)* trigger a restart. This can be quite helpful if you are iterating on a feature that uses a cloud service that you don't have locally. Generally remote updates and restarts are much quicker than a full rebuild and deploy cycle.

> **Note**
>
> Files are only monitored when the remote client is running. If you change a file before starting the remote client, it won't be pushed to the remote server.

Remote debug tunnel

Java remote debugging is useful when diagnosing issues on a remote application. Unfortunately, it's not always possible to enable remote debugging when your application is deployed outside of your data center. Remote debugging can also be tricky to setup if you are using a container based technology such as Docker.

To help work around these limitations, devtools supports tunneling of remote debug traffic over HTTP. The remote client provides a local server on port `8000` that you can attach a remote debugger to. Once a connection is established, debug traffic is sent over HTTP to the remote application. You can use the `spring.devtools.remote.debug.local-port` property if you want to use a different port.

You'll need to ensure that your remote application is started with remote debugging enabled. Often this can be achieved by configuring `JAVA_OPTS`. For example, with Cloud Foundry you can add the following to your `manifest.yml`:

```
---
  env:
    JAVA_OPTS: "-Xdebug -Xrunjdwp:server=y,transport=dt_socket,suspend=n"
```

> **Tip**
>
> Notice that you don't need to pass an `address=NNNN` option to `-Xrunjdwp`. If omitted Java will simply pick a random free port.

> **Note**
>
> Debugging a remote service over the Internet can be slow and you might need to increase timeouts in your IDE. For example, in Eclipse you can select `Java` → `Debug` from `Preferences…` and change the `Debugger timeout (ms)` to a more suitable value (`60000` works well in most situations).

21. Packaging your application for production

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

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

22. What to read next

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.

23. SpringApplication

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 ::   v1.4.0.M2

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.

23.1 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.charset` (default is `UTF-8`). In addition a text file, you can also add a `banner.gif`, `banner.jpg` or `banner.png` image file to your classpath, or set a `banner.image.location` property. Images will be converted into an ASCII art representation and printed above any text banner.

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

Table 23.1. Banner variables

Variable	Description
`${application.version}`	The version number of your application as declared in `MANIFEST.MF`. For example `Implementation-Version: 1.0` is printed as `1.0`.
`${application.formatted-version}`	The version number of your application as declared in `MANIFEST.MF` formatted for display (surrounded with brackets and prefixed with v). For example `(v1.0)`.

Variable	Description
`${spring-boot.version}`	The Spring Boot version that you are using. For example `1.4.0.M2`.
`${spring-boot.formatted-version}`	The Spring Boot version that you are using formatted for display (surrounded with brackets and prefixed with v). For example `(v1.4.0.M2)`.
`${Ansi.NAME}` (or `${AnsiColor.NAME}`, `${AnsiBackground.NAME}`, `${AnsiStyle.NAME}`)	Where `NAME` is the name of an ANSI escape code. See `AnsiPropertySource` for details.
`${application.title}`	The title of your application as declared in `MANIFEST.MF`. For example `Implementation-Title: MyApp` is printed as `MyApp`.

Tip

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

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

The printed banner will be registered as a singleton bean under the name `springBootBanner`.

Note

YAML maps `off` to `false` so make sure to add quotes if you want to disable the banner in your application.

```
spring:
    main:
        banner-mode: "off"
```

23.2 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.setBannerMode(Banner.Mode.OFF);
    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 24, Externalized Configuration* for details.

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

23.3 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()
    .bannerMode(Banner.Mode.OFF)
    .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.

23.4 Application events and listeners

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

> **Note**
>
> Some events are actually triggered before the `ApplicationContext` is created so you cannot register a listener on those as a `@Bean`. You can register them via the `SpringApplication.addListeners(…)` or `SpringApplicationBuilder.listeners(…)` methods.
>
> If you want those listeners to be registered automatically regardless of the way the application is created you can add a `META-INF/spring.factories` file to your project and reference your listener(s) using the `org.springframework.context.ApplicationListener` key.
>
> ```
> org.springframework.context.ApplicationListener=com.example.project.MyListener
> ```

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 `ApplicationReadyEvent` is sent after the refresh and any related callbacks have been processed to indicate the application is ready to service requests.
5. 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.

23.5 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.

23.6 Accessing application arguments

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

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

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

}
```

Tip

Spring Boot will also register a `CommandLinePropertySource` with the Spring `Environment`. This allows you to also inject single application arguments using the `@Value` annotation.

23.7 Using the ApplicationRunner or CommandLineRunner

If you need to run some specific code once the `SpringApplication` has started, you can implement the `ApplicationRunner` or `CommandLineRunner` interfaces. Both interfaces work in the same way and offer a single `run` method which will be called just before `SpringApplication.run(…)` completes.

The `CommandLineRunner` interfaces provides access to application arguments as a simple string array, whereas the `ApplicationRunner` uses the `ApplicationArguments` interface discussed above.

```
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` or `ApplicationRunner` beans are defined that must be called in a specific order.

23.8 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.

23.9 Admin features

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

Tip

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

Note

Take care when enabling this feature as the MBean exposes a method to shutdown the application.

24. Externalized Configuration

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 via `@ConfigurationProperties`.

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

1. Command line arguments.
2. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property)
3. JNDI attributes from `java:comp/env`.
4. Java System properties (`System.getProperties()`).
5. OS environment variables.
6. A `RandomValuePropertySource` that only has properties in `random.*`.
7. Profile-specific application properties outside of your packaged jar (`application-{profile}.properties` and YAML variants)
8. Profile-specific application properties packaged inside your jar (`application-{profile}.properties` and YAML variants)
9. Application properties outside of your packaged jar (`application.properties` and YAML variants).
10. Application properties packaged inside your jar (`application.properties` and YAML variants).
11. `@PropertySource` annotations on your `@Configuration` classes.
12. 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;

    // ...

}
```

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

Tip

The `SPRING_APPLICATION_JSON` properties can be supplied on the command line with an environment variable. For example in a UN*X shell:

```
$ SPRING_APPLICATION_JSON='{"foo":{"bar":"spam"}}' java -jar myapp.jar
```

In this example you will end up with `foo.bar=spam` in the Spring `Environment`. You can also supply the JSON as `spring.application.json` in a System variable:

```
$ java -Dspring.application.json='{"foo":"bar"}' -jar myapp.jar
```

or command line argument:

```
$ java -jar myapp.jar --spring.application.json='{"foo":"bar"}'
```

or as a JNDI variable `java:comp/env/spring.application.json`.

24.1 Configuring random values

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).

24.2 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)`.

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

The list is ordered by precedence (properties defined in locations higher in the list override those defined in lower locations).

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
```

Warning

`spring.config.name` and `spring.config.location` are used very early to determine which files have to be loaded so they have to be defined as an environment property (typically OS env, system property or command line argument).

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, including profile-specific file names). Files specified in `spring.config.location` are used as-is, with no support for profile-specific variants, and will be overridden by any profile-specific properties.

The default search path `classpath:,classpath:/config,file:,file:config/` is always used, irrespective of the value of `spring.config.location`. This search path is ordered from lowest to highest precedence (`file:config/` wins). If you do specify your own locations, they take precedence over all of the default locations and use the same lowest to highest precedence ordering. 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.

24.4 Profile-specific properties

In addition to `application.properties` files, profile-specific properties can also be defined using the naming convention `application-{profile}.properties`. The `Environment` has a set of default profiles (by default `[default]`) which are used if no active profiles are set (i.e. if no profiles are explicitly activated then properties from `application-default.properties` are loaded).

Profile-specific properties are loaded from the same locations as standard `application.properties`, with profile-specific files always overriding the non-specific ones irrespective of whether the profile-specific files are inside or outside your packaged jar.

If several profiles are specified, a last wins strategy applies. For example, profiles specified by the `spring.profiles.active` property are added after those configured via the `SpringApplication` API and therefore take precedence.

Note

If you have specified any files in `spring.config.location`, profile-specific variants of those files will not be considered. Use directories in`spring.config.location` if you also want to also use profile-specific properties.

24.5 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 69.4, “Use ‘short’ command line arguments”* how-to for details.

24.6 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`.

The default profiles are activated if none are explicitly active when the application context starts. So in this YAML we set a value for `security.user.password` that is **only** available in the "default" profile:

```
server:
  port: 8000
---
spring:
  profiles: default
security:
  user:
    password: weak
```

whereas in this example, the password is always set because it isn't attached to any profile, and it would have to be explicitly reset in all other profiles as necessary:

```
server:
  port: 8000
security:
  user:
    password: weak
```

Spring profiles designated using the "spring.profiles" element may optionally be negated using the {@code !} character. If both negated and non-negated profiles are specified for a single document, at least one non-negated profile must match and no negated profiles may match.

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.

24.7 Type-safe Configuration Properties

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

}
```

Note

The getters and setters are advisable, since binding is via standard Java Beans property descriptors, just like in Spring MVC. They are mandatory for immutable types or those that are directly coercible from `String`. As long as they are initialized, maps, collections, and arrays need a getter but not necessarily a setter since they can be mutated by the binder. If there is a setter, Maps, collections, and arrays can be created. Maps and collections can be expanded with only a getter, whereas arrays require a setter. Nested POJO properties can also be created (so a setter is not mandatory) if they have a default constructor, or a constructor accepting a single value that can be coerced from String. Some people use Project Lombok to add getters and setters automatically.

Note

Contrary to `@Value`, SpEL expressions are not evaluated prior to injecting a value in the relevant `@ConfigurationProperties` bean.

The `@EnableConfigurationProperties` annotation is automatically applied to your project so that any beans annotated with `@ConfigurationProperties` will be 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 {
}
```

Note

When `@ConfigurationProperties` bean is registered that way, the bean will have a conventional name: `<prefix>-<fqn>`, where `<prefix>` is the environment key prefix specified in the `@ConfigurationProperties` annotation and <fqn> the fully qualified name of the bean. If the annotation does not provide any prefix, only the fully qualified name of the bean is used.

> The bean name in the example above will be `connection-com.example.ConnectionSettings`, assuming that `ConnectionSettings` sits in the `com.example` package.

> **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.

Third-party configuration

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

To configure a bean from the `Environment` properties, add `@ConfigurationProperties` to its bean registration:

```
@ConfigurationProperties(prefix = "foo")
@Bean
public FooComponent fooComponent() {
    ...
}
```

Any property defined with the `foo` prefix will be mapped onto that `FooComponent` bean in a similar manner as the `ConnectionSettings` example above.

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 dashed separated (e.g. `context-path` binds to `contextPath`), and capitalized (e.g. `PORT` binds to `port`) environment properties.

For example, given the following `@ConfigurationProperties` class:

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

    private String firstName;

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

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

}
```

The following properties names can all be used:

Table 24.1. relaxed binding

Property	Note
`person.firstName`	Standard camel case syntax.
`person.first-name`	Dashed notation, recommended for use in `.properties` and `.yml` files.
`person.first_name`	Underscore notation, alternative format for use in `.properties` and `.yml` files.
`PERSON_FIRST_NAME`	Upper case format. Recommended when using a system environment variables.

Properties conversion

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) or custom `Converters` (with bean definitions annotated as `@ConfigurationPropertiesBinding`).

Note

As this bean is requested very early during the application lifecycle, make sure to limit the dependencies that your `ConversionService` is using. Typically, any dependency that you require may not be fully initialized at creation time. You may want to rename your custom `ConversionService` if it's not required for configuration keys coercion and only rely on custom converters qualified with `@ConfigurationPropertiesBinding`.

@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

}
```

In order to validate values of nested properties, you must annotate the associated field as `@Valid` to trigger its validation. For example, building upon the above `ConnectionSettings` example:

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

    @NotNull
    @Valid
    private RemoteAddress remoteAddress;
    // ... getters and setters

    public static class RemoteAddress {

        @NotEmpty
        public String hostname;

        // ... getters and setters

    }

}
```

You can also add a custom Spring `Validator` by creating a bean definition called `configurationPropertiesValidator`. There is a Validation sample so you can see how to set things up.

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.

25. Profiles

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`.

25.1 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 69.7, “Change configuration depending on the environment” for more details.

25.2 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.

25.3 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 24.4, “Profile-specific properties”* for details.

26. Logging

Spring Boot uses Commons Logging for all internal logging, but leaves the underlying log implementation open. Default configurations are provided for Java Util Logging, Log4J2 and Logback. In each case loggers are pre-configured to use console output with optional file output also available.

By default, If you use the ‘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.

26.1 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.
- Thread name — Enclosed in square brackets (may be truncated for console output).
- Logger name — This is usually the source class name (often abbreviated).
- The log message.

> **Note**
>
> Logback does not have a `FATAL` level (it is mapped to `ERROR`)

26.2 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. You can also enable a “debug” mode by starting your application with a `--debug` flag.

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

> **Note**
>
> you can also specify `debug=true` in your `application.properties`.

When the debug mode is enabled, a selection of core loggers (embedded container, Hibernate and Spring Boot) are configured to output more information. Enabling the debug mode does *not* configure your application to log all messages with `DEBUG` level.

Alternatively, you can enable a “trace” mode by starting your application with a `--trace` flag (or `trace=true` in your `application.properties`). This will enable trace logging for a selection of core loggers (embedded container, Hibernate schema generation and the whole Spring portfolio).

Color-coded output

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

Color coding is configured using the `%clr` conversion word. In its simplest form the converter will color the output according to the log level, for example:

```
%clr(%5p)
```

The mapping of log level to a color is as follows:

Level	**Color**
`FATAL`	Red
`ERROR`	Red
`WARN`	Yellow
`INFO`	Green
`DEBUG`	Green
`TRACE`	Green

Alternatively, you can specify the color or style that should be used by providing it as an option to the conversion. For example, to make the text yellow:

```
%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){yellow}
```

The following colors and styles are supported:

- `blue`
- `cyan`
- `faint`
- `green`
- `magenta`

- `red`
- `yellow`

26.3 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 a `logging.file` or `logging.path` property (for example in your `application.properties`).

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

Table 26.1. Logging properties

`logging.file`	**`logging.path`**	**Example**	**Description**
(none)	*(none)*		Console only logging.
Specific file	*(none)*	`my.log`	Writes to the specified log file. Names can be an exact location or relative to the current directory.
(none)	Specific directory	`/var/log`	Writes `spring.log` to the specified directory. Names can be an exact location or relative to the current directory.

Log files will rotate when they reach 10 Mb and as with console output, `ERROR`, `WARN` and `INFO` level messages are logged by default.

Note

The logging system is initialized early in the application lifecycle and as such logging properties will not be found in property files loaded via `@PropertySource` annotations.

Tip

Logging properties are independent of the actual logging infrastructure. As a result, specific configuration keys (such as `logback.configurationFile` for Logback) are not managed by spring Boot.

26.4 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. The `root` logger can be configured using `logging.level.root`. Example `application.properties`:

```
logging.level.root=WARN
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

Note

By default Spring Boot remaps Thymeleaf `INFO` messages so that they are logged at `DEBUG` level. This helps to reduce noise in the standard log output. See `LevelRemappingAppender` for details of how you can apply remapping in your own configuration.

26.5 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`.

You can force Spring Boot to use a particular logging system using the `org.springframework.boot.logging.LoggingSystem` system property. The value should be the fully-qualified class name of a `LoggingSystem` implementation. You can also disable Spring Boot's logging configuration entirely by using a value of `none`.

Note

Since logging is initialized **before** the `ApplicationContext` is created, it 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-spring.xml`, `logback-spring.groovy`, `logback.xml` or `logback.groovy`
Log4j2	`log4j2-spring.xml` or `log4j2.xml`
JDK (Java Util Logging)	`logging.properties`

Note

When possible we recommend that you use the `-spring` variants for your logging configuration (for example `logback-spring.xml` rather than `logback.xml`). If you use standard configuration locations, Spring cannot completely control log initialization.

Warning

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

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

Spring Environment	System Property	Comments
`logging.exception-conversion-word`	`LOG_EXCEPTION_CONVERSION_WORD`	The conversion word that's used when logging exceptions.
`logging.file`	`LOG_FILE`	Used in default log configuration if defined.

Spring Environment	System Property	Comments
`logging.path`	`LOG_PATH`	Used in default log configuration if defined.
`logging.pattern.console`	`CONSOLE_LOG_PATTERN`	The log pattern to use on the console (stdout). (Not supported with JDK logger.)
`logging.pattern.file`	`FILE_LOG_PATTERN`	The log pattern to use in a file (if LOG_FILE enabled). (Not supported with JDK logger.)
`logging.pattern.level`	`LOG_LEVEL_PATTERN`	The format to use to render the log level (default `%5p`). (The `logging.pattern.level` form is only supported by Logback.)
`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.

Tip

If you want to use a placeholder in a logging property, you should use Spring Boot's syntax and not the syntax of the underlying framework. Notably, if you're using Logback, you should use `:` as the delimiter between a property name and its default value and not `:-`.

Tip

You can add MDC and other ad-hoc content to log lines by overriding only the `LOG_LEVEL_PATTERN` (or `logging.pattern.level` with Logback). For example, if you use `logging.pattern.level=user:%X{user} %5p` then the default log format will contain an MDC entry for "user" if it exists, e.g.

```
2015-09-30 12:30:04.031 user:juergen INFO 22174 --- [  nio-8080-exec-0] demo.Controller Handling
 authenticated request
```

26.6 Logback extensions

Spring Boot includes a number of extensions to Logback which can help with advanced configuration. You can use these extensions in your `logback-spring.xml` configuration file.

Note

You cannot use extensions in the standard `logback.xml` configuration file since it's loaded too early. You need to either use `logback-spring.xml` or define a `logging.config` property.

Profile-specific configuration

The `<springProfile>` tag allows you to optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the `<configuration>` element. Use the `name` attribute to specify which profile accepts the configuration. Multiple profiles can be specified using a comma-separated list.

```
<springProfile name="staging">
    <!-- configuration to be enabled when the "staging" profile is active -->
</springProfile>

<springProfile name="dev, staging">
    <!-- configuration to be enabled when the "dev" or "staging" profiles are active -->
</springProfile>

<springProfile name="!production">
    <!-- configuration to be enabled when the "production" profile is not active -->
</springProfile>
```

Environment properties

The `<springProperty>` tag allows you to surface properties from the Spring `Environment` for use within Logback. This can be useful if you want to access values from your `application.properties` file in your logback configuration. The tag works in a similar way to Logback's standard `<property>` tag, but rather than specifying a direct `value` you specify the `source` of the property (from the `Environment`). You can use the `scope` attribute if you need to store the property somewhere other than in `local` scope. If you need a fallback value in case the property is not set in the `Environment`, you can use the `defaultValue` attribute.

```
<springProperty scope="context" name="fluentHost" source="myapp.fluentd.host"
        defaultValue="localhost"/>
<appender name="FLUENT" class="ch.qos.logback.more.appenders.DataFluentAppender">
    <remoteHost>${fluentHost}</remoteHost>
    ...
</appender>
```

Tip

The `RelaxedPropertyResolver` is used to access `Environment` properties. If specify the `source` in dashed notation (`my-property-name`) all the relaxed variations will be tried (`myPropertyName`, `MY_PROPERTY_NAME` etc).

27. Developing web applications

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.

27.1 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 `MessageCodesResolver` (see below).
- Static `index.html` support.

- Custom `Favicon` support.
- Automatic use of a `ConfigurableWebBindingInitializer` bean (see below).

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.

Custom JSON Serializers and Deserializers

If you're using Jackson to serialize and deserialize JSON data, you might want to write your own `JsonSerializer` and `JsonDeserializer` classes. Custom serializers are usually registered with Jackson via a Module, but Spring Boot provides an alternative `@JsonComponent` annotation which makes it easier to directly register Spring Beans.

You can use `@JsonComponent` directly on `JsonSerializer` or `JsonDeserializer` implementations. You can also use it on classes that contains serializers/deserializers as inner-classes. For example:

```
import java.io.*;
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import org.springframework.boot.jackson.*;

@JsonComponent
public class Example {

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

}
```

All `@JsonComponent` beans in the `ApplicationContext` will be automatically registered with Jackson, and since `@JsonComponent` is meta-annotated with `@Component`, the usual component-scanning rules apply.

Spring Boot also provides `JsonObjectSerializer` and `JsonObjectDeserializer` base classes which provide useful alternatives to the standard Jackson versions when serializing Objects. See the Javadoc for details.

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 directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath or from the root of the `ServletContext`. It uses the `ResourceHttpRequestHandler` from Spring MVC so 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`.

You can customize the static resource locations using `spring.resources.staticLocations` (replacing the default values with a list of directory locations). If you do this the default welcome page detection will switch to your custom locations, so if there is an `index.html` in any of your locations on startup, it will be the home page of the application.

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` directory if your application will be packaged as a jar. Although this directory 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.

Spring Boot also supports advanced resource handling features provided by Spring MVC, allowing use cases such as cache busting static resources or using version agnostic URLs for Webjars.

For example, the following configuration will configure a cache busting solution for all static resources, effectively adding a content hash in URLs, such as `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`:

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

Note

Links to resources are rewritten at runtime in template, thanks to a `ResourceUrlEncodingFilter`, auto-configured for Thymeleaf, Velocity and FreeMarker. You should manually declare this filter when using JSPs. Other template engines aren't automatically supported right now, but can be with custom template macros/helpers and the use of the `ResourceUrlProvider`.

When loading resources dynamically with, for example, a JavaScript module loader, renaming files is not an option. That's why other strategies are also supported and can be combined. A "fixed" strategy will add a static version string in the URL, without changing the file name:

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
spring.resources.chain.strategy.fixed.enabled=true
spring.resources.chain.strategy.fixed.paths=/js/lib/
spring.resources.chain.strategy.fixed.version=v12
```

With this configuration, JavaScript modules located under `"/js/lib/"` will use a fixed versioning strategy `"/v12/js/lib/mymodule.js"` while other resources will still use the content one `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`.

See `ResourceProperties` for more of the supported options.

Tip

This feature has been thoroughly described in a dedicated blog post and in Spring Framework's reference documentation.

ConfigurableWebBindingInitializer

Spring MVC uses a `WebBindingInitializer` to initialize a `WebDataBinder` for a particular request. If you create your own `ConfigurableWebBindingInitializer` `@Bean`, Spring Boot will automatically configure Spring MVC to use it.

Template engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies including 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 (deprecated in 1.4)

- Mustache

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

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**
>
> IntelliJ IDEA orders the classpath differently depending on how you run your application. Running your application in the IDE via its main method will result in a different ordering to when you run your application using Maven or Gradle or from its packaged jar. This can cause Spring Boot to fail to find the templates on the classpath. If you're affected by this problem you can reorder the classpath in the IDE to place the module's classes and resources first. Alternatively, you can configure the template prefix to search every templates directory on the classpath: `classpath*:/templates/`.

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.

> **Tip**
>
> The `BasicErrorController` can be used as a base class for a custom `ErrorController`. This is particularly useful if you want to add a handler for a new content type (the default is to handle `text/html` specifically and provide a fallback for everything else). To do that just extend `BasicErrorController` and add a public method with a `@RequestMapping` that has a `produces` attribute, and create a bean of your new type.

You can also define a `@ControllerAdvice` to customize the JSON document to return for a particular controller and/or exception type.

```
@ControllerAdvice(basePackageClasses = FooController.class)
public class FooControllerAdvice extends ResponseEntityExceptionHandler {

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

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

}
```

In the example above, if `YourException` is thrown by a controller defined in the same package as `FooController`, a json representation of the `CustomerErrorType` POJO will be used instead of the `ErrorAttributes` representation.

If you want more specific error pages for some conditions, the embedded servlet containers support a uniform Java DSL for customizing the error handling. Assuming that you have a mapping for `/400`:

```
@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`

Spring HATEOAS

If you're developing a RESTful API that makes use of hypermedia, Spring Boot provides auto-configuration for Spring HATEOAS that works well with most applications. The auto-configuration replaces the need to use `@EnableHypermediaSupport` and registers a number of beans to ease building hypermedia-based applications including a `LinkDiscoverers` (for client side support) and an `ObjectMapper` configured to correctly marshal responses into the desired representation. The `ObjectMapper` will be customized based on the `spring.jackson.*` properties or a `Jackson2ObjectMapperBuilder` bean if one exists.

You can take control of Spring HATEOAS's configuration by using `@EnableHypermediaSupport`. Note that this will disable the `ObjectMapper` customization described above.

CORS support

Cross-origin resource sharing (CORS) is a W3C specification implemented by most browsers that allows you to specify in a flexible way what kind of cross domain requests are authorized, instead of using some less secure and less powerful approaches like IFRAME or JSONP.

As of version 4.2, Spring MVC supports CORS out of the box. Using controller method CORS configuration with `@CrossOrigin` annotations in your Spring Boot application does not require any specific configuration. Global CORS configuration can be defined by registering a `WebMvcConfigurer` bean with a customized `addCorsMappings(CorsRegistry)` method:

```
@Configuration
public class MyConfiguration {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurerAdapter() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**");
            }
        };
    }
}
```

27.2 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 CXF 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);
    }

}
```

You can also register an arbitrary number of beans implementing `ResourceConfigCustomizer` for more advanced customizations.

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`. By default, the servlet will be initialized lazily but you can customize it with `spring.jersey.servlet.load-on-startup` .You can disable or override that bean by creating one of your own with the same name. You can also use a Filter instead of a Servlet by setting `spring.jersey.type=filter` (in which case the `@Bean` to replace or override is `jerseyFilterRegistration`). The 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 packaged as nested jars.

27.3 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, Filters, and listeners

When using an embedded servlet container you can register Servlets, Filters and all the listeners from the Servlet spec (e.g. `HttpSessionListener`) either by using Spring beans or by scanning for Servlet components.

Registering Servlets, Filters, and listeners as Spring beans

Any `Servlet`, `Filter` or Servlet `*Listener` instance that is a Spring bean will be registered with the embedded container. This can be particularly convenient if you want to refer to a value from your `application.properties` during configuration.

By default, if the context contains only a single Servlet it 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`, `FilterRegistrationBean` and `ServletListenerRegistrationBean` classes for complete control.

Servlet Context Initialization

Embedded servlet containers will not directly execute the Servlet 3.0+ `javax.servlet.ServletContainerInitializer` interface, or Spring's `org.springframework.web.WebApplicationInitializer` interface. This is an intentional design decision intended to reduce the risk that 3rd party libraries designed to run inside a war will break Spring Boot applications.

If you need to perform servlet context initialization in a Spring Boot application, you should register a bean that implements the `org.springframework.boot.context.embedded.ServletContextInitializer` interface. The single `onStartup` method provides access to the `ServletContext`, and can easily be used as an adapter to an existing `WebApplicationInitializer` if necessary.

Scanning for Servlets, Filters, and listeners

When using an embedded container, automatic registration of `@WebServlet`, `@WebFilter`, and `@WebListener` annotated classes can be enabled using `@ServletComponentScan`.

> **Tip**
>
> `@ServletComponentScan` will have no effect in a standalone container, where the container's built-in discovery mechanisms will be used instead.

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:

- Network settings: listen port for incoming HTTP requests (`server.port`), interface address to bind to `server.address`, etc.
- Session settings: whether the session is persistent (`server.session.persistence`), session timeout (`server.session.timeout`), location of session data (`server.session.store-dir`) and session-cookie configuration (`server.session.cookie.*`).

- Error management: location of the error page (`server.error.path`), etc.
- SSL
- HTTP compression

Spring Boot tries as much as possible to expose common settings but this is not always possible. For those cases, dedicated namespaces offer server-specific customizations (see `server.tomcat` and `server.undertow`). For instance, access logs can be configured with specific features of the embedded servlet container.

Tip

See the `ServerProperties` class for a complete list.

Programmatic customization

If you need to configure your embedded 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.NOT_FOUND, "/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.

28. Security

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
```

Note

If you fine-tune your logging configuration, ensure that the `org.springframework.boot.autoconfigure.security` category is set to log `INFO` messages, otherwise the default password will not be printed.

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 default web security configuration completely you can add a bean with `@EnableWebSecurity` (this does not disable the authentication manager configuration). To customize it you normally use external properties and beans of type `WebSecurityConfigurerAdapter` (e.g. to add form-based login). To also switch off the authentication manager configuration you can add a bean of type `AuthenticationManager`, or else configure the global `AuthenticationManager` by autowiring an `AuthenticationManagerBuilder` into a method in one of your `@Configuration` classes. 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 (insecure) 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 auto-configured features add a `@Bean` of type `WebSecurityConfigurerAdapter` with `@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)`.

28.1 OAuth2

If you have `spring-security-oauth2` on your classpath you can take advantage of some auto-configuration to make it easy to set up Authorization or Resource Server.

Authorization Server

To create an Authorization Server and grant access tokens you need to use `@EnableAuthorizationServer` and provide `security.oauth2.client.client-id` and `security.oauth2.client.client-secret]` properties. The client will be registered for you in an in-memory repository.

Having done that you will be able to use the client credentials to create an access token, for example:

```
$ curl client:secret@localhost:8080/oauth/token -d grant_type=password -d username=user -d password=pwd
```

The basic auth credentials for the `/token` endpoint are the `client-id` and `client-secret`. The user credentials are the normal Spring Security user details (which default in Spring Boot to “user” and a random password).

To switch off the auto-configuration and configure the Authorization Server features yourself just add a `@Bean` of type `AuthorizationServerConfigurer`.

Resource Server

To use the access token you need a Resource Server (which can be the same as the Authorization Server). Creating a Resource Server is easy, just add `@EnableResourceServer` and provide some configuration to allow the server to decode access tokens. If your application is also an Authorization Server it already knows how to decode tokens, so there is nothing else to do. If your app is a standalone service then you need to give it some more configuration, one of the following options:

- `security.oauth2.resource.user-info-uri` to use the `/me` resource (e.g. `https://uaa.run.pivotal.io/userinfo` on PWS)
- `security.oauth2.resource.token-info-uri` to use the token decoding endpoint (e.g. `https://uaa.run.pivotal.io/check_token` on PWS).

If you specify both the `user-info-uri` and the `token-info-uri` then you can set a flag to say that one is preferred over the other (`prefer-token-info=true` is the default).

Alternatively (instead of `user-info-uri` or `token-info-uri`) if the tokens are JWTs you can configure a `security.oauth2.resource.jwt.key-value` to decode them locally (where the key is a verification key). The verification key value is either a symmetric secret or PEM-encoded RSA public key. If you don’t have the key and it’s public you can provide a URI where it can be downloaded (as a JSON object with a “value” field) with `security.oauth2.resource.jwt.key-uri`. E.g. on PWS:

```
$ curl https://uaa.run.pivotal.io/token_key
{"alg":"SHA256withRSA","value":"-----BEGIN PUBLIC KEY-----\nMIIBI...\n-----END PUBLIC KEY-----\n"}
```

Warning

If you use the `security.oauth2.resource.jwt.key-uri` the authorization server needs to be running when your application starts up. It will log a warning if it can’t find the key, and tell you what to do to fix it.

28.2 Token Type in User Info

Google, and certain other 3rd party identity providers, are more strict about the token type name that is sent in the headers to the user info endpoint. The default is “Bearer” which suits most providers and matches the spec, but if you need to change it you can set `security.oauth2.resource.token-type`.

28.3 Customizing the User Info RestTemplate

If you have a `user-info-uri`, the resource server features use an `OAuth2RestTemplate` internally to fetch user details for authentication. This is provided as a qualified `@Bean` with id `userInfoRestTemplate`, but you shouldn’t need to know that to just use it. The default should be fine for most providers, but occasionally you might need to add additional interceptors, or change the request authenticator (which is how the token gets attached to outgoing requests). To add a customization just create a bean of type `UserInfoRestTemplateCustomizer` - it has a single method that will be called after the bean is created but before it is initialized. The rest template that is being customized here is *only* used internally to carry out authentication.

Tip

To set an RSA key value in YAML use the “pipe” continuation marker to split it over multiple lines (“|”) and remember to indent the key value (it’s a standard YAML language feature). Example:

```
security:
    oauth2:
        resource:
            jwt:
                keyValue: |
                    -----BEGIN PUBLIC KEY-----
                    MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC...
                    -----END PUBLIC KEY-----
```

Client

To make your webapp into an OAuth2 client you can simply add `@EnableOAuth2Client` and Spring Boot will create an `OAuth2RestTemplate` for you to `@Autowire`. It uses the `security.oauth2.client.*` as credentials (the same as you might be using in the Authorization Server), but in addition it will need to know the authorization and token URIs in the Authorization Server. For example:

application.yml.

```
security:
    oauth2:
        client:
            clientId: bd1c0a783ccdd1c9b9e4
            clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1
            accessTokenUri: https://github.com/login/oauth/access_token
            userAuthorizationUri: https://github.com/login/oauth/authorize
            clientAuthenticationScheme: form
```

An application with this configuration will redirect to Github for authorization when you attempt to use the `OAuth2RestTemplate`. If you are already signed into Github you won’t even notice that it has authenticated. These specific credentials will only work if your application is running on port 8080 (register your own client app in Github or other provider for more flexibility).

To limit the scope that the client asks for when it obtains an access token you can set `security.oauth2.client.scope` (comma separated or an array in YAML). By default the scope is empty and it is up to Authorization Server to decide what the defaults should be, usually depending on the settings in the client registration that it holds.

> **Note**
>
> There is also a setting for `security.oauth2.client.client-authentication-scheme` which defaults to “header” (but you might need to set it to “form” if, like Github for instance, your OAuth2 provider doesn’t like header authentication). In fact, the `security.oauth2.client.*` properties are bound to an instance of `AuthorizationCodeResourceDetails` so all its properties can be specified.

> **Tip**
>
> In a non-web application you can still `@Autowire` an `OAuth2RestOperations` and it is still wired into the `security.oauth2.client.*` configuration. In this case it is a “client credentials token grant” you will be asking for if you use it (and there is no need to use `@EnableOAuth2Client` or `@EnableOAuth2Sso`). To switch it off, just remove the `security.oauth2.client.client-id` from your configuration (or make it the empty string).

Single Sign On

An OAuth2 Client can be used to fetch user details from the provider (if such features are available) and then convert them into an `Authentication` token for Spring Security. The Resource Server above support this via the `user-info-uri` property This is the basis for a Single Sign On (SSO) protocol based on OAuth2, and Spring Boot makes it easy to participate by providing an annotation `@EnableOAuth2Sso`. The Github client above can protect all its resources and authenticate using the Github `/user/` endpoint, by adding that annotation and declaring where to find the endpoint (in addition to the `security.oauth2.client.*` configuration already listed above):

application.yml.

```yaml
security:
    oauth2:
...
    resource:
        userInfoUri: https://api.github.com/user
        preferTokenInfo: false
```

Since all paths are secure by default, there is no “home” page that you can show to unauthenticated users and invite them to login (by visiting the `/login` path, or the path specified by `security.oauth2.sso.login-path`).

To customize the access rules or paths to protect, so you can add a “home” page for instance, `@EnableOAuth2Sso` can be added to a `WebSecurityConfigurerAdapter` and the annotation will cause it to be decorated and enhanced with the necessary pieces to get the `/login` path working. For example, here we simply allow unauthenticated access to the home page at "/" and keep the default for everything else:

```java
@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    public void init(WebSecurity web) {
        web.ignore("/");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/**").authorizeRequests().anyRequest().authenticated();
    }

}
```

28.4 Actuator Security

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

- The management endpoints are secure even if the application endpoints are insecure.
- 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.

29. Working with SQL databases

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

29.1 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`.

Tip

If, for whatever reason, you do configure the connection URL for an embedded database, care should be taken to ensure that the database's automatic shutdown is disabled. If you're using H2 you should use `DB_CLOSE_ON_EXIT=FALSE` to do so. If you're using HSQLDB, you should ensure that `shutdown=true` is not used. Disabling the database's automatic shutdown allows Spring Boot to control when the database is closed, thereby ensuring that it happens once access to the database is no longer needed.

Connection to a production database

Production database connections can also be auto-configured 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 automatically get a dependency to `tomcat-jdbc`.

Note

You can bypass that algorithm completely and specify the connection pool to use via the `spring.datasource.type` property. Also, 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
```

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.driver-class-name=com.mysql.jdbc.Driver` then that class has to be loadable.

See `DataSourceProperties` for more of the supported options. These are the standard options that work regardless of the actual implementation. It is also possible to fine-tune implementation-specific settings using their respective prefix (`spring.datasource.tomcat.*`, `spring.datasource.hikari.*`, `spring.datasource.dbcp.*` and `spring.datasource.dbcp2.*`). Refer to the documentation of the connection pool implementation you are using for more details.

For instance, if you are using the Tomcat connection pool you could customize many additional settings:

```
# Number of ms to wait before throwing an exception if no connection is available.
spring.datasource.tomcat.max-wait=10000
```

```
# Maximum number of active connections that can be allocated from this pool at the same time.
spring.datasource.tomcat.max-active=50

# Validate the connection before borrowing it from the pool.
spring.datasource.tomcat.test-on-borrow=true
```

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
```

29.2 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;
    }

    // ...

}
```

29.3 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 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 73.4, “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.

29.4 Using H2's web console

The H2 database provides a browser-based console that Spring Boot can auto-configure for you. The console will be auto-configured when the following conditions are met:

- You are developing a web application

- `com.h2database:h2` is on the classpath
- You are using Spring Boot’s developer tools

Tip

If you are not using Spring Boot’s developer tools, but would still like to make use of H2’s console, then you can do so by configuring the `spring.h2.console.enabled` property with a value of `true`. The H2 console is only intended for use during development so care should be taken to ensure that `spring.h2.console.enabled` is not set to `true` in production.

Changing the H2 console’s path

By default the console will be available at `/h2-console`. You can customize the console’s path using the `spring.h2.console.path` property.

Securing the H2 console

When Spring Security is on the classpath and basic auth is enabled, the H2 console will be automatically secured using basic auth. The following properties can be used to customize the security configuration:

- `security.user.role`
- `security.basic.authorize-mode`
- `security.basic.enabled`

30. Using jOOQ

Java Object Oriented Querying (jOOQ) is a popular product from Data Geekery which generates Java code from your database, and lets you build type safe SQL queries through its fluent API. Both the commercial and open source editions can be used with Spring Boot.

30.1 Code Generation

In order to use jOOQ type-safe queries, you need to generate Java classes from your database schema. You can follow the instructions in the jOOQ user manual. If you are using the `jooq-codegen-maven` plugin (and you also use the `spring-boot-starter-parent` "parent POM") you can safely omit the plugin's `<version>` tag. You can also use Spring Boot defined version variables (e.g. `h2.version`) to declare the plugin's database dependency. Here's an example:

```
<plugin>
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <executions>
        ...
    </executions>
    <dependencies>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>${h2.version}</version>
        </dependency>
    </dependencies>
    <configuration>
        <jdbc>
            <driver>org.h2.Driver</driver>
            <url>jdbc:h2:~/yourdatabase</url>
        </jdbc>
        <generator>
            ...
        </generator>
    </configuration>
</plugin>
```

30.2 Using DSLContext

The fluent API offered by jOOQ is initiated via the `org.jooq.DSLContext` interface. Spring Boot will auto-configure a `DSLContext` as a Spring Bean and connect it to your application `DataSource`. To use the `DSLContext` you can just `@Autowire` it:

```
@Component
public class JooqExample implements CommandLineRunner {

    private final DSLContext create;

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

}
```

Tip

The jOOQ manual tends to use a variable named `create` to hold the `DSLContext`, we've done the same for this example.

You can then use the `DSLContext` to construct your queries:

```
public List<GregorianCalendar> authorsBornAfter1980() {
    return this.create.selectFrom(AUTHOR)
        .where(AUTHOR.DATE_OF_BIRTH.greaterThan(new GregorianCalendar(1980, 0, 1)))
        .fetch(AUTHOR.DATE_OF_BIRTH);
}
```

30.3 Customizing jOOQ

You can customize the SQL dialect used by jOOQ by setting `spring.jooq.sql-dialect` in your `application.properties`. For example, to specify Postgres you would add:

```
spring.jooq.sql-dialect=Postgres
```

More advanced customizations can be achieved by defining your own `@Bean` definitions which will be used when the jOOQ `Configuration` is created. You can define beans for the following jOOQ Types:

- `ConnectionProvider`
- `TransactionProvider`
- `RecordMapperProvider`
- `RecordListenerProvider`
- `ExecuteListenerProvider`
- `VisitListenerProvider`

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

31. Working with NoSQL technologies

Spring Data provides additional projects that help you access a variety of NoSQL technologies including MongoDB, Neo4J, Elasticsearch, Solr, Redis, Gemfire, Couchbase and Cassandra. Spring Boot provides auto-configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr and Cassandra; 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.

31.1 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-data-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.

31.2 MongoDB

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

Connecting to a MongoDB database

You can inject an auto-configured `org.springframework.data.mongodb.MongoDbFactory` to access Mongo 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 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 and configure additional settings such as the *replica set*:

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

Alternatively, as long as you're using Mongo 2.x, 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
```

Note

`spring.data.mongodb.host` and `spring.data.mongodb.port` are not supported if you're using the Mongo 3.0 Java driver. In such cases, `spring.data.mongodb.uri` should be used to provide all of the configuration.

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` bean 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.

Embedded Mongo

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

The port that Mongo will listen on can be configured using the `spring.data.mongodb.port` property. To use a randomly allocated free port use a value of zero. The `MongoClient` created by `MongoAutoConfiguration` will be automatically configured to use the randomly allocated port.

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

You can declare your own `IMongodConfig` and `IRuntimeConfig` beans to take control of the Mongo instance's configuration and logging routing.

31.3 Neo4j

Neo4j is an open-source NoSQL graph database that uses a rich data model of nodes related by first class relationships which is better suited for connected big data than traditional rdbms approaches. Spring Boot offers several conveniences for working with Neo4j, including the `spring-boot-starter-data-neo4j` ‘Starter POM’.

Connecting to a Neo4j database

You can inject an auto-configured `Neo4jSession`, `Session` or `Neo4jOperations` instance as you would any other Spring Bean. By default the instance will attempt to connect to a Neo4j server using `localhost:7474`:

```
@Component
public class MyBean {

    private final Neo4jTemplate neo4jTemplate;

    @Autowired
    public MyBean(Neo4jTemplate neo4jTemplate) {
        this.neo4jTemplate = neo4jTemplate;
    }

    // ...

}
```

You can take full control of the configuration by adding a `org.neo4j.ogm.config.Configuration` `@Bean` of your own. Also, adding a `@Bean` of type `Neo4jOperations` disables the auto-configuration.

You can configure the user and credentials to use via the `spring.data.neo4j.*` properties:

```
spring.data.neo4j.uri=http://my-server:7474
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=secret
```

Using the embedded mode

Note

Neo4j’s embedded mode is subject to a different licensing, make sure to review it before integrating the dependency in your application.

If you add `org.neo4j:neo4j-ogm-embedded-driver` to the dependencies of your application, Spring Boot will automatically configure an in-process embedded instance of Neo4j that will not persist any data when your application shuts down. You can explicitly disable that mode using `spring.data.neo4j.embedded.enabled=false`. You can also enable persistence for the embedded mode:

```
spring.data.neo4j.uri=file://var/tmp/graph.db
```

Neo4jSession

By default, the lifetime of the session is scope to the application. If you are running a web application you can change it to scope or request easily:

```
spring.data.neo4j.session.scope=session
```

Spring Data Neo4j repositories

Spring Data includes repository support for Neo4j.

In fact, both Spring Data JPA and Spring Data Neo4j share the same common infrastructure; so you could take the JPA example from earlier and, assuming that `City` is now a Neo4j OGM `@NodeEntity` rather than a JPA `@Entity`, it will work in the same way.

Tip

You can customize entity scanning locations using the `@NodeEntityScan` annotation.

To enable repository support (and optionally support for `@Transactional`), add the following two annotations to your Spring configuration:

```
@EnableNeo4jRepositories(basePackages = "com.example.myapp.repository")
@EnableTransactionManagement
```

Repository example

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

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

public interface CityRepository extends GraphRepository<City> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndCountry(String name, String country);

}
```

Tip

For complete details of Spring Data Neo4j, including its rich object mapping technologies, refer to their reference documentation.

31.4 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-configuration support for Gemfire, but you can enable Spring Data Repositories with a single annotation (`@EnableGemfireRepositories`).

31.5 Solr

Apache Solr is a search engine. Spring Boot offers basic auto-configuration for the Solr 5 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 `SolrClient` 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 SolrClient solr;
    @Autowired
    public MyBean(SolrClient solr) {
        this.solr = solr;
    }

    // ...

}
```

If you add a `@Bean` of your own of type `SolrClient` 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.

31.6 Elasticsearch

Elasticsearch 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 embed a local in-memory server (a `Node` in ElasticSearch terms) and use the current working directory as the home directory for the server. In this setup, the first thing to do is to tell ElasticSearch were to store its files:

```
spring.data.elasticsearch.properties.path.home=/foo/bar
```

Alternatively, you can switch to a remote server (i.e. a `TransportClient`) by setting `spring.data.elasticsearch.cluster-nodes` to a comma-separated 'host:port' list.

```
spring.data.elasticsearch.cluster-nodes=localhost:9300
```

```
@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.

31.7 Cassandra

Cassandra is an open source, distributed database management system designed to handle large amounts of data across many commodity servers. Spring Boot offers auto-configuration for Cassandra and abstractions on top of it provided by Spring Data Cassandra. There is a `spring-boot-starter-data-cassandra` ‘Starter POM’ for collecting the dependencies in a convenient way.

Connecting to Cassandra

You can inject an auto-configured `CassandraTemplate` or a Cassandra `Session` instance as you would with any other Spring Bean. The `spring.data.cassandra.*` properties can be used to customize the connection. Generally you will provide `keyspace-name` and `contact-points` properties:

```
spring.data.cassandra.keyspace-name=mykeyspace
spring.data.cassandra.contact-points=cassandrahost1,cassandrahost2
```

```
@Component
public class MyBean {

    private CassandraTemplate template;

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

    // ...

}
```

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

Spring Data Cassandra repositories

Spring Data includes basic repository support for Cassandra. Currently this is more limited than the JPA repositories discussed earlier, and will need to annotate finder methods with `@Query`.

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

31.8 Couchbase

Couchbase is an open-source, distributed multi-model NoSQL document-oriented database that is optimized for interactive applications. Spring Boot offers auto-configuration for Couchbase and abstractions on top of it provided by Spring Data Couchbase. There is a `spring-boot-starter-data-couchbase` ‘Starter POM’ for collecting the dependencies in a convenient way.

Connecting to Couchbase

You can very easily get a `Bucket` and `Cluster` by adding the Couchbase SDK and some configuration. The `spring.couchbase.*` properties can be used to customize the connection. Generally you will provide the bootstrap hosts, bucket name and password:

```
spring.couchbase.bootstrap-hosts=my-host-1,192.168.1.123
spring.couchbase.bucket.name=my-bucket
spring.couchbase.bucket.password=secret
```

Tip

You need to provide *at least* the bootstrap host(s), in which case the bucket name is `default` and the password is the empty String. Alternatively, you can define your own `org.springframework.data.couchbase.config.CouchbaseConfigurer` `@Bean` to take control over the whole configuration.

It is also possible to customize some of the `CouchbaseEnvironment` settings. For instance the following configuration changes the timeout to use to open a new `Bucket` and enable SSL support:

```
spring.couchbase.env.timeouts.connect=3000
spring.couchbase.env.ssl.key-store=/location/of/keystore.jks
spring.couchbase.env.ssl.key-store-password=secret
```

Check the `spring.couchbase.env.*` properties for more details.

Spring Data Couchbase repositories

Spring Data includes repository support for Couchbase. For complete details of Spring Data Couchbase, refer to their reference documentation.

You can inject an auto-configured `CouchbaseTemplate` instance as you would with any other Spring Bean as long as a *default* `CouchbaseConfigurer` is available (that happens when you enable the couchbase support as explained above). If you want to bypass the auto-configuration for Spring Data Couchbase, provide your own `org.springframework.data.couchbase.config.AbstractCouchbaseDataConfiguration` implementation.

```
@Component
public class MyBean {

    private final CouchbaseTemplate template;

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

    // ...
}
```

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

32. Caching

The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to methods, reducing thus the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker.

Note

Check the relevant section of the Spring Framework reference for more details.

In a nutshell, adding caching to an operation of your service is as easy as adding the relevant annotation to its method:

```
import javax.cache.annotation.CacheResult;

import org.springframework.stereotype.Component;

@Component
public class MathService {

    @CacheResult
    public int computePiDecimal(int i) {
        // ...
    }

}
```

Note

You can either use the standard JSR-107 (JCache) annotations or Spring's own caching annotations transparently. We strongly advise you however to not mix and match them.

Tip

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

32.1 Supported cache providers

The cache abstraction does not provide an actual store and relies on abstraction materialized by the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces. Spring Boot auto-configures a suitable `CacheManager` according to the implementation as long as the caching support is enabled via the `@EnableCaching` annotation.

Note

If you are using the cache infrastructure with beans that are not interface-based, make sure to enable the `proxyTargetClass` attribute of `@EnableCaching`.

Tip

Use the `spring-boot-starter-cache` “Starter POM” to quickly add required caching dependencies. If you are adding dependencies manually you should note that certain implementations are only provided by the `spring-context-support` jar.

If you haven't defined a bean of type `CacheManager` or a `CacheResolver` named `cacheResolver` (see `CachingConfigurer`), Spring Boot tries to detect the following providers (in this order):

- Generic
- JCache (JSR-107)
- EhCache 2.x
- Hazelcast
- Infinispan
- Couchbase
- Redis
- Caffeine
- Guava
- Simple

It is also possible to *force* the cache provider to use via the `spring.cache.type` property.

If the `CacheManager` is auto-configured by Spring Boot, you can further tune its configuration before it is fully initialized by exposing a bean implementing the `CacheManagerCustomizer` interface. The following sets the cache names to use.

```
@Bean
public CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer() {
    return new CacheManagerCustomizer<ConcurrentMapCacheManager>() {
        @Override
        public void customize(ConcurrentMapCacheManager cacheManager) {
            cacheManager.setCacheNames(Arrays.asList("one", "two"));
        }
    };
}
```

Note

=== In the example above, a `ConcurrentMapCacheManager` is expected to be configured. If that is not the case, the customizer won't be invoked at all. You can have as many customizers as you want and you can also order them as usual using `@Order` or `Ordered`. ===

Generic

Generic caching is used if the context defines *at least* one `org.springframework.cache.Cache` bean, a `CacheManager` wrapping them is configured.

JCache

JCache is bootstrapped via the presence of a `javax.cache.spi.CachingProvider` on the classpath (i.e. a JSR-107 compliant caching library). It might happen that more than one provider is present, in which case the provider must be explicitly specified. Even if the JSR-107 standard does not enforce a standardized way to define the location of the configuration file, Spring Boot does its best to accommodate with implementation details.

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

Note

Since a cache library may offer both a native implementation and JSR-107 support Spring Boot will prefer the JSR-107 support so that the same features are available if you switch to a different JSR-107 implementation.

There are several ways to customize the underlying `javax.cache.cacheManager`:

- Caches can be created on startup via the `spring.cache.cache-names` property. If a custom `javax.cache.configuration.Configuration` bean is defined, it is used to customize them.
- `org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer` beans are invoked with the reference of the `CacheManager` for full customization.

Tip

If a standard `javax.cache.CacheManager` bean is defined, it is wrapped automatically in a `org.springframework.cache.CacheManager` implementation that the abstraction expects. No further customization is applied on it.

EhCache 2.x

EhCache 2.x is used if a file named `ehcache.xml` can be found at the root of the classpath. If EhCache 2.x and such file is present it is used to bootstrap the cache manager. An alternate configuration file can be provide a well using:

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

Hazelcast

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

If for some reason you need a different `HazelcastInstance` for caching, you can request Spring Boot to create a separate one that will be only used by the `CacheManager`:

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

Tip

If a separate `HazelcastInstance` is created that way, it is not registered in the application context.

Infinispan

Infinispan has no default configuration file location so it must be specified explicitly (or the default bootstrap is used).

```
spring.cache.infinispan.config=infinispan.xml
```

Caches can be created on startup via the `spring.cache.cache-names` property. If a custom `ConfigurationBuilder` bean is defined, it is used to customize them.

Couchbase

If Couchbase is available and configured, a `CouchbaseCacheManager` is auto-configured. It is also possible to create additional caches on startup using the `spring.cache.cache-names` property. These will operate on the `Bucket` that was auto-configured. You can *also* create additional caches on another `Bucket` using the customizer: assume you need two caches on the "main" `Bucket` (`foo` and `bar`) and one `biz` cache with a custom time to live of 2sec on the `another` `Bucket`. First, you can create the two first caches simply via configuration:

```
spring.cache.cache-names=foo,bar
```

Then define this extra `@Configuration` to configure the extra `Bucket` and the `biz` cache:

```
@Configuration
public class CouchbaseCacheConfiguration {

    private final Cluster cluster;

    public CouchbaseCacheConfiguration(Cluster cluster) {
        this.cluster = cluster;
    }

    @Bean
    public Bucket anotherBucket() {
        return this.cluster.openBucket("another", "secret");
    }

    @Bean
    public CacheManagerCustomizer<CouchbaseCacheManager> cacheManagerCustomizer() {
        return c -> {
            c.prepareCache("biz", CacheBuilder.newInstance(anotherBucket())
                    .withExpirationInMillis(2000));
        };
    }

}
```

This sample configuration reuses the `Cluster` that was created via auto-configuration.

Redis

If Redis is available and configured, the `RedisCacheManager` is auto-configured. It is also possible to create additional caches on startup using the `spring.cache.cache-names` property.

Note

By default, a key prefix is added to prevent that if two separate caches use the same key, Redis would have overlapping keys and be likely to return invalid values. We strongly recommend to keep this setting enabled if you create your own `RedisCacheManager`.

Caffeine

Caffeine is a Java 8 rewrite of Guava’s cache and will supersede the Guava support in Spring Boot 2.0. If Caffeine is present, a `CaffeineCacheManager` is auto-configured. Caches can be created on startup using the `spring.cache.cache-names` property and customized by one of the following (in this order):

1. A cache spec defined by `spring.cache.caffeine.spec`
2. A `com.github.benmanes.caffeine.cache.CaffeineSpec` bean is defined
3. A `com.github.benmanes.caffeine.cache.Caffeine` bean is defined

For instance, the following configuration creates a `foo` and `bar` caches with a maximum size of 500 and a *time to live* of 10 minutes

```
spring.cache.cache-names=foo,bar
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s
```

Besides, if a `com.github.benmanes.caffeine.cache.CacheLoader` bean is defined, it is automatically associated to the `CaffeineCacheManager`.

Guava

If Guava is present, a `GuavaCacheManager` is auto-configured. Caches can be created on startup using the `spring.cache.cache-names` property and customized by one of the following (in this order):

1. A cache spec defined by `spring.cache.guava.spec`
2. A `com.google.common.cache.CacheBuilderSpec` bean is defined
3. A `com.google.common.cache.CacheBuilder` bean is defined

For instance, the following configuration creates a `foo` and `bar` caches with a maximum size of 500 and a *time to live* of 10 minutes

```
spring.cache.cache-names=foo,bar
spring.cache.guava.spec=maximumSize=500,expireAfterAccess=600s
```

Besides, if a `com.google.common.cache.CacheLoader` bean is defined, it is automatically associated to the `GuavaCacheManager`.

Simple

If none of these options worked out, a simple implementation using `ConcurrentHashMap` as cache store is configured. This is the default if no caching library is present in your application.

33. Messaging

The Spring Framework provides extensive support for integrating with messaging systems: from simplified use of the JMS API using `JmsTemplate` to a complete infrastructure to receive messages asynchronously. Spring AMQP provides a similar feature set for the 'Advanced Message Queuing Protocol' and Spring 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.

33.1 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.

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.

Artemis support

Apache Artemis was formed in 2015 when HornetQ was donated to the Apache Foundation. All the features listed in the the section called "HornetQ support" section below can be applied to Artemis. Simply replace `spring.hornetq.*` properties with `spring.artemis.*` and use `spring-boot-starter-artemis` instead of `spring-boot-starter-hornetq`. If you want to embed Artemis, make sure to add `org.apache.activemq:artemis-jms-server` to the dependencies of your application.

Note

You should not try and use Artemis and HornetQ and the same time.

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 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 choose 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.

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 can be injected in a similar manner. If a `DestinationResolver` or `MessageConverter` beans are defined, they are associated automatically to the auto-configured `JmsTemplate`.

Receiving a message

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

The default factory is transactional by default. If you are running in an infrastructure where a `JtaTransactionManager` is present, it will be associated to the listener container by default. If not, the `sessionTransacted` flag will be enabled. In that latter scenario, you can associate your local data store transaction to the processing of an incoming message by adding `@Transactional` on your listener method (or a delegate thereof). This will make sure that the incoming message is acknowledged once the local transaction has completed. This also includes sending response messages that have been performed on the same JMS session.

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

```
@Component
public class MyBean {

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

}
```

Tip

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

If you need to create more `JmsListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `DefaultJmsListenerContainerFactoryConfigurer` that you can use to initialize a `DefaultJmsListenerContainerFactory` with the same settings as the one that is auto-configured.

For instance, the following exposes another factory that uses a specific `MessageConverter`:

```
@Configuration
static class JmsConfiguration {

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

}
```

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

```
@Component
public class MyBean {

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

}
```

33.2 AMQP

The Advanced Message Queuing Protocol (AMQP) is a platform-neutral, wire-level protocol for message-oriented middleware. The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions.

RabbitMQ support

RabbitMQ is a lightweight, reliable, scalable and portable message broker based on the AMQP protocol. Spring uses `RabbitMQ` to communicate using the AMQP protocol.

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

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

See `RabbitProperties` for more of the supported options.

Tip

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

Sending a message

Spring’s `AmqpTemplate` and `AmqpAdmin` are auto-configured and you can autowire them directly into your own beans:

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final AmqpAdmin amqpAdmin;
    private final AmqpTemplate amqpTemplate;

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

    // ...

}
```

Note

RabbitMessagingTemplate can be injected in a similar manner. If a MessageConverter bean is defined, it is associated automatically to the auto-configured AmqpTemplate.

Any org.springframework.amqp.core.Queue that is defined as a bean will be automatically used to declare a corresponding queue on the RabbitMQ instance if necessary.

You can enable retries on the AmqpTemplate to retry operations, for example in the event the broker connection is lost. Retries are disabled by default.

Receiving a message

When the Rabbit infrastructure is present, any bean can be annotated with @RabbitListener to create a listener endpoint. If no RabbitListenerContainerFactory has been defined, a default one is configured automatically. If a MessageConverter beans is defined, it is associated automatically to the default factory.

The following component creates a listener endpoint on the someQueue queue:

```
@Component
public class MyBean {

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

}
```

Tip

Check the Javadoc of @EnableRabbit for more details.

If you need to create more RabbitListenerContainerFactory instances or if you want to override the default, Spring Boot provides a SimpleRabbitListenerContainerFactoryConfigurer that you can use to initialize a SimpleRabbitListenerContainerFactory with the same settings as the one that is auto-configured.

For instance, the following exposes another factory that uses a specific MessageConverter:

```
@Configuration
static class RabbitConfiguration {

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

}
```

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

```
@Component
public class MyBean {

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

}
```

You can enable retries to handle situations where your listener throws an exception. When retries are exhausted, the message will be rejected and either dropped or routed to a dead-letter exchange if the broker is configured so. Retries are disabled by default.

Important

If retries are not enabled and the listener throws an exception, by default the delivery will be retried indefinitely. You can modify this behavior in two ways; set the `defaultRequeueRejected` property to `false` and zero redeliveries will be attempted; or, throw an `AmqpRejectAndDontRequeueException` to signal the message should be rejected. This is the mechanism used when retries are enabled and the maximum delivery attempts are reached.

34. Sending email

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.

35. Distributed Transactions with JTA

Spring Boot supports distributed JTA transactions across multiple XA resources using either an Atomikos 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.

35.1 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` directory 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.atomikos.properties` can also be used to customize the Atomikos `UserTransactionServiceImp`. 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.

35.2 Using a Bitronix transaction manager

Bitronix is popular open source JTA transaction manager implementation. You can use the `spring-boot-starter-jta-bitronix` starter POM to add the appropriate Bitronix 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` directory in your application home directory. You can customize this directory by using the `spring.jta.log-dir` property. Properties starting `spring.jta.bitronix.properties` 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.

35.3 Using a Narayana transaction manager

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

By default Narayana transaction logs will be written to a `transaction-logs` directory 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.narayana.properties` can also be used to customize the Narayana configuration. See the `NarayanaProperties` Javadoc for complete details.

Note

To ensure that multiple transaction managers can safely coordinate the same resource managers, each Narayana instance must be configured with a unique ID. By default this ID is set to `1`. 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.

35.4 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`.

35.5 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:

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

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

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

35.6 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.

36. Hazelcast

If hazelcast is on the classpath, Spring Boot will auto-configure an `HazelcastInstance` that you can inject in your application. The `HazelcastInstance` is only created if a configuration is found.

You can define a `com.hazelcast.config.Config` bean and we'll use that. If your configuration defines an instance name, we'll try to locate an existing instance rather than creating a new one.

You could also specify the `hazelcast.xml` configuration file to use via configuration:

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

Otherwise, Spring Boot tries to find the Hazelcast configuration from the default locations, that is `hazelcast.xml` in the working directory or at the root of the classpath. We also check if the `hazelcast.config` system property is set. Check the Hazelcast documentation for more details.

Note

Spring Boot also has an explicit caching support for Hazelcast. The `HazelcastInstance` is automatically wrapped in a `CacheManager` implementation if caching is enabled.

37. Spring Integration

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.

38. Spring Session

Spring Session provides support for managing a user’s session information. If you are writing a web application and Spring Session and Spring Data Redis are both on the classpath, Spring Boot will auto-configure Spring Session through its `@EnableRedisHttpSession`. Session data will be stored in Redis and the session timeout can be configured using the `server.session.timeout` property.

39. Monitoring and management over JMX

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.

40. Testing

Spring Boot provides a number of utilities and annotations to help when testing your application. Test support is provided by two modules; `spring-boot-test` contains core items, and `spring-boot-test-autoconfigure` supports auto-configuration for tests.

Most developers will just use the the `spring-boot-starter-test` ‘Starter POM’ which imports both Spring Boot test modules as well has JUnit, AssertJ, Hamcrest and a number of other useful libraries.

40.1 Test scope dependencies

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

- JUnit — The de-facto standard for unit testing Java applications.
- Spring Test & Spring Boot Test — utilities and integration test support for Spring Boot applications.
- AssertJ - A fluent assertion library.
- Hamcrest — A library of matcher objects (also known as constraints or predicates).
- Mockito — A Java mocking framework.
- JSONassert — An assertion library for JSON.
- JsonPath — XPath for JSON.

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.

40.2 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.

40.3 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 `@SpringBootTest` annotation which can be used as an alternative the standard `spring-test` `@ContextConfiguration` annotation when you need Spring Boot features. The annotation works by creating the `ApplicationContext` used in your tests via `SpringApplication`.

You can use the `webEnvironment` attribute of `@SpringBootTest` to further refine how your tests will run:

- `MOCK` — Loads a `WebApplicationContext` and provides a mock servlet environment. Embedded servlet containers are not started when using this annotation. If servlet APIs are not on your classpath this mode will transparently fallback to creating a regular non-web `ApplicationContext`.
- `RANDOM_PORT` — Loads an `EmbeddedWebApplicationContext` and provides a real servlet environment. Embedded servlet containers are started and listening on a random port.
- `DEFINED_PORT` — Loads an `EmbeddedWebApplicationContext` and provides a real servlet environment. Embedded servlet containers are started and listening on a defined port (i.e from your `application.properties` or on the default port `8080`).
- `NONE` — Loads an `ApplicationContext` using `SpringApplication` but does not provides *any* servlet environment (mock or otherwise).

> **Note**
>
> In addition to `@SpringBootTest` a number of other annotations are also provided for testing more specific slices of an application. See below for details.

> **Tip**
>
> Don't forget to also add `@RunWith(SpringRunner.class)` to your test, otherwise the annotations will be ignored.

Detecting test configuration

If you're familiar with the Spring Test Framework, you may be used to using `@ContextConfiguration(classes=…)` in order to specify which Spring `@Configuration` to load. Alternatively, you might have often used nested `@Configuration` classes within your test.

When testing Spring Boot applications this is often not required. Spring Boot's `@*Test` annotations will search for your primary configuration automatically whenever you don't explicitly define one.

The search algorithm works up from the package that contains the test until it finds a `@SpringBootApplication` or `@SpringBootConfiguration` annotated class. As long as you've structure your code in a sensible way your main configuration is usually found.

If you want to customize the primary configuration, you can use a nested `@TestConfiguration` class. Unlike a nested `@Configuration` class which would be used instead of a your application's primary configuration, a nested `@TestConfiguration` class will be used in addition to your application's primary configuration.

> **Note**
>
> Spring's test framework will cache application contexts between tests. Therefore, as long as your tests share the same configuration (no matter how it's discovered), the potentially time consuming process of loading the context will only happen once.

Excluding test configuration

If your application uses component scanning, for example if you use `@SpringBootApplication` or `@ComponentScan`, you may find components or configurations created only for specific tests accidentally get picked up everywhere.

To help prevent this, Spring Boot provides `@TestComponent` and `@TestConfiguration` annotations that can be used on classes in `src/test/java` to indicate that they should not be picked up by scanning.

Note

`@TestComponent` and `@TestConfiguration` are only needed on top level classes. If you define `@Configuration` or `@Component` as inner-classes within a test, they will be automatically filtered.

Note

If you directly use `@ComponentScan` (i.e. not via `@SpringBootApplication`) you will need to register the `TypeExcludeFilter` with it. See the Javadoc for details.

Working with random ports

If you need to start a full running server for tests, we recommend that you use random ports. If you use `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)` an available port will be picked at random each time your test runs.

The `@LocalServerPort` annotation can be used to inject the actual port used into your test. For convenience, tests that need to make REST calls to the started server can additionally `@Autowire` a `TestRestTemplate` which will resolve relative links to the running server.

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.context.web.*;
import org.springframework.boot.test.web.client.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class MyWebIntegrationTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void exampleTest() {
        String body = this.restTemplate.getForObject("/", String.class);
        assertThat(body).isEqualTo("Hello World");
    }

}
```

Mocking and spying beans

It’s sometimes necessary to mock certain components within your application context when running tests. For example, you may have a facade over some remote service that’s unavailable during development. Mocking can also be useful when you want to simulate failures that might be hard to trigger in a real environment.

Spring Boot includes a `@MockBean` annotation that can be used to define a Mockito mock for a bean inside your `ApplicationContext`. You can use the annotation to add new beans, or replace a single existing bean definition. The annotation can be used directly on test classes, on fields within your test; or on `@Configuration` classes and fields. When used on a field the, instance of the created mock will also be injected. Mock beans are automatically reset after each test method.

Here's a typical example where we replace an existing `RemoteService` bean with a mock implementation:

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.mock.mockito.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Reverser reverser;

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

}
```

Additionally you can also use `@SpyBean` to wrap any existing bean with a Mockito `spy`. See the javadoc for full details.

Auto-configured tests

Spring Boot's auto-configuration system works well for applications, but can sometimes be a little too much for tests. It's often helpful to load only the parts of the configuration that are required to test a 'slice' of your application. For example, you might want to test that Spring MVC controllers are mapping URLs correctly, and you don't want to involve and database calls in those tests; or you *might be wanting* to test JPA entities, and you're not interested in web layer when those tests run.

The `spring-boot-test-autoconfigure` module includes a number of annotations that can be used to automatically configure such 'slices'. Each of them work in a similar way, providing a `@…Test` annotation that loads the `ApplicationContext` and one or more `@AutoConfigure…` annotations that can be used to customize auto-configuration settings.

Tip

It's also possible to use the `@AutoConfigure…` annotations with the standard `@SpringBootTest` annotation. You can use this combination if you're not interested in 'slicing' your application but you want some of the auto-configured test beans.

Auto-configured JSON tests

To test that Object JSON serialization and deserialization is working as expected you can use the `@JsonTest` annotation. `@JsonTest` will auto-configure Jackson ObjectMappers, any `@JsonComponent` beans and any Jackson `Modules`. It also configures `Gson` if you happen to be using that instead of, or as well as, Jackson. If you need to configure elements of the auto-configuration you can use the `@AutoConfigureJsonTesters` annotation.

Spring Boot includes AssertJ based helpers that work with the JSONassert and JsonPath libraries to check that JSON is as expected. The `JacksonHelper`, `GsonHelper` and `BasicJsonHelper` classes can be used for Jackson, Gson and Strings respectively. Any helper fields on the test class will be automatically initialized when using `@JsonTest`.

```java
import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.autoconfigure.json.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.json.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@JsonTest
public class MyJsonTests {

    private JacksonTester<VehicleDetails> json;

    @Test
    public void testSerialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda", "Civic");
        // Assert against a `.json` file in the same package as the test
        assertThat(this.json.write(details)).isEqualToJson("expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");
        assertThat(this.json.write(details)).extractingJsonPathStringValue("@.make")
                .isEqualTo("Honda");
    }

    @Test
    public void testDeserialize() throws Exception {
        String content = "{\"make\":\"Ford\",\"model\":\"Focus\"}";
        assertThat(this.json.parse(content))
                .isEqualTo(new VehicleDetails("Ford", "Focus"));
        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
    }

}
```

Note

JSON helper classes can also be used directly in standard unit tests. Simply call the `initFields` method of the helper in your `@Before` method if you aren't using `@JsonTest`.

Auto-configured Spring MVC tests

To test Spring MVC controllers are working as expected you can use the `@WebMvcTest` annotation. `@WebMvcTest` will auto-configure the Spring MVC infrastructure and limit scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Filter`, `WebMvcConfigurer` and `HandlerMethodArgumentResolver`. Regular `@Component` beans will not be scanned when using this annotation.

Often `@WebMvcTest` will be limited to a single controller and used in combination with `@MockBean` to provide mock implementations for required collaborators.

`@WebMvcTest` is meta-annotated with `@AutoConfigureMockMvc` which provides auto-configuration of `MockMvc`. Mock MVC offers a powerful way to quickly test MVC controllers without needing to start a full HTTP server.

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    public void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
                .willReturn(new VehicleDetails("Honda", "Civic"));
        this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
                .andExpect(status().isOk()).andExpect(content().string("Honda Civic"));
    }

}
```

Tip

If you need to configure elements of the auto-configuration (for example when servlet filters should be applied) you can use attributes in the `@AutoConfigureMockMvc` annotation.

If you use HtmlUnit or Selenium, auto-configuration will also provide a `WebClient` bean and/or a `WebDriver` bean. Here is an example that uses HtmlUnit:

```
import com.gargoylesoftware.htmlunit.*;
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;
```

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyHtmlUnitTests {

    @Autowired
    private WebClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    public void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
                .willReturn(new VehicleDetails("Honda", "Civic"));
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
        assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
    }

}
```

Auto-configured Data JPA tests

The `@DataJpaTest` can be used if want to test JPA applications. By default it will configure an in-memory embedded database, scan for `@Entity` classes and configure Spring Data JPA repositories. Regular `@Component` beans will not be loaded into the `ApplicationContext`.

Data JPA tests may also inject a `TestEntityManager` bean which provides an alternative to the standard JPA `EntityManager` specifically designed for tests. If you want to use `TestEntityManager` outside of `@DataJpaTests` you can also use the `@AutoConfigureTestEntityManager` annotation.

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.autoconfigure.orm.jpa.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

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

}
```

In-memory embedded databases generally work well for tests since they are fast and don't require any developer installation. If, however, you prefer to run tests against a real database you can use the `@AutoConfigureTestDatabase` annotation:

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
```

```
public class ExampleRepositoryTests {

    // ...

}
```

Auto-configured Spring REST Docs tests

Test `@AutoConfigureRestDocs` annotation can be used if you want to use Spring REST Docs in your tests. It will automatically configure `MockMvc` to use Spring REST Docs and removes the need for Spring REST Docs' JUnit rule.

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
@AutoConfigureRestDocs("target/generated-snippets")
public class UserDocumentationTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void listUsers() throws Exception {
        this.mvc.perform(get("/users").accept(MediaType.TEXT_PLAIN))
                .andExpect(status().isOk())
                .andDo(document("list-users"));
    }

}
```

In addition to configuring the output directory, `@AutoConfigureRestDocs` can also configure the host, scheme, and port that will appear in any documented URIs. If you require more control over Spring REST Docs' configuration a `RestDocsMockMvcConfigurationCustomizer` bean can be used:

```
@TestConfiguration
static class CustomizationConfiguration
        implements RestDocsMockMvcConfigurationCustomizer {

    @Override
    public void customize(MockMvcRestDocumentationConfigurer configurer) {
        configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
    }

}
```

If you want to make use of Spring REST Docs' support for a parameterized output directory, you can create a `RestDocumentationResultHandler` bean. The auto-configuration will call `alwaysDo` with this result handler, thereby causing each `MockMvc` call to automatically generate the default snippets:

```
@TestConfiguration
static class ResultHandlerConfiguration{

    @Bean
    public RestDocumentationResultHandler restDocumentation() {
        return MockMvcRestDocumentation.document("{method-name}");
    }

}
```

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.

Note

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

40.4 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 `@SpringBootTest`.

```
@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.rule.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"));
 }

}
```

41. Creating your own auto-configuration

If you work in a company that develops shared libraries, or if you work on an open-source or commercial library, you might want to develop your own auto-configuration. Auto-configuration classes can be bundled in external jars and still be picked-up by Spring Boot.

Auto-configuration can be associated to a "starter" that provides the auto-configuration code as well as the typical libraries that you would use with it. We will first cover what you need to know to build your own auto-configuration and we will move on to the typical steps required to create a custom starter.

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

41.1 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).

41.2 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`.

If you want to order certain auto-configurations that shouldn't have any direct knowledge of each other, you can also use `@AutoconfigureOrder`. That annotation has the same semantic as the regular `@Order` annotation but provides a dedicated order for auto-configuration classes.

> **Note**
>
> Auto-configurations have to be loaded that way *only*. Make sure that they are defined in a specific package space and that they are never the target of component scan in particular.

41.3 Condition annotations

You almost always want to include one or more `@Conditional` 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 included 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 a bean to be included based on the presence or absence of specific beans. You can use the `value` attribute to specify beans by type, or `name` to specify beans by name. The `search` attribute allows you to limit the `ApplicationContext` hierarchy that should be considered when searching for beans.

> **Tip**
>
> You need to be very careful about the order that bean definitions are added as these conditions are evaluated based on what has been processed so far. For this reason, we recommend only using `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations on auto-configuration classes (since these are guaranteed to load after any user-define beans definitions have been added).

> **Note**
>
> `@ConditionalOnBean` and `@ConditionalOnMissingBean` do not prevent `@Configuration` classes from being created. Using these conditions at the class level is equivalent to marking each contained `@Bean` method with the annotation.

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 included 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 included based on the result of a SpEL expression.

41.4 Creating your own starter

A full Spring Boot starter for a library may contain the following components:

- The `autoconfigure` module that contains the auto-configuration code.
- The `starter` module that provides a dependency to the autoconfigure module as well as the library and any additional dependencies that are typically useful. In a nutshell, adding the starter should be enough to start using that library.

> **Tip**
>
> You may combine the auto-configuration code and the dependency management in a single module if you don't need to separate those two concerns.

Naming

Please make sure to provide a proper namespace for your starter. Do not start your module names with `spring-boot`, even if you are using a different Maven groupId. We may offer an official support for the thing you're auto-configuring in the future.

Here is a rule of thumb. Let's assume that you are creating a starter for "acme", name the auto-configure module `acme-spring-boot-autoconfigure` and the starter `acme-spring-boot-starter`. If you only have one module combining the two, use `acme-spring-boot-starter`.

Besides, if your starter provides configuration keys, use a proper namespace for them. In particular, do not include your keys in the namespaces that Spring Boot uses (e.g. `server`, `management`, `spring`, etc). These are "ours" and we may improve/modify them in the future in such a way it could break your things.

Make sure to trigger meta-data generation so that IDE assistance is available for your keys as well. You may want to review the generated meta-data (`META-INF/spring-configuration-metadata.json`) to make sure your keys are properly documented.

Autoconfigure module

The autoconfigure module contains everything that is necessary to get started with the library. It may also contain configuration keys definition (`@ConfigurationProperties`) and any callback interface that can be used to further customize how the components are initialized.

> **Tip**
>
> You should mark the dependencies to the library as optional so that you can include the autoconfigure module in your projects more easily. If you do it that way, the library won't be provided and Spring Boot will back off by default.

Starter module

The starter is an empty jar, really. Its only purpose is to provide the necessary dependencies to work with the library; see it as an opinionated view of what is required to get started.

Do not make assumptions about the project in which your starter is added. If the library you are auto-configuring typically requires other starters, mention them as well. Providing a proper set of *default* dependencies may be hard if the number of optional dependencies is high as you should avoid bringing unnecessary dependencies for a typical usage of the library.

42. WebSockets

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.

43. What to read next

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

If you are comfortable with Spring Boot's core features, you can 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.

Actuator HTTP endpoints are only available with a Spring MVC-based application. In particular, it will not work with Jersey unless you enable Spring MVC as well.

44. Enabling production-ready features

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")
}
```

45. Endpoints

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 Default
`actuator`	Provides a hypermedia-based “discovery page” for the other endpoints. Requires Spring HATEOAS to be on the classpath.	true
`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
`docs`	Displays documentation, including example requests and responses, for the Actuator’s endpoints. Requires `spring-boot-actuator-docs` to be on the classpath.	false
`dump`	Performs a thread dump.	true
`env`	Exposes properties from Spring’s `ConfigurableEnvironment`.	true
`flyway`	Shows any Flyway database migrations that have been applied.	true
`health`	Shows application health information (when the application is secure, a simple ‘status’ when accessed over an unauthenticated connection or full message details when authenticated).	false
`info`	Displays arbitrary application info.	false
`liquibase`	Shows any Liquibase database migrations that have been applied.	true
`logfile`	Returns the contents of the logfile (if `logging.file` or `logging.path` properties have been set). Only available via MVC. Supports the use of the HTTP `Range` header to retrieve part of the log file’s content.	true

ID	Description	Sensitive Default
`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` property 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).

45.1 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
```

Likewise, you can also choose to globally set the “sensitive” flag of all endpoints. By default, the sensitive flag depends on the type of endpoint (see the table above). For example, to mark *all* endpoints as sensitive except `info`:

```
endpoints.sensitive=true
endpoints.info.sensitive=false
```

45.2 Hypermedia for actuator MVC endpoints

If Spring HATEOAS is on the classpath (e.g. through the `spring-boot-starter-hateoas` or if you are using Spring Data REST) then the HTTP endpoints from the Actuator are enhanced with hypermedia links, and a “discovery page” is added with links to all the endpoints. The “discovery page” is available on `/actuator` by default. It is implemented as an endpoint, allowing properties to be used to configure its path (`endpoints.actuator.path`) and whether or not it is enabled (`endpoints.actuator.enabled`).

When a custom management context path is configured, the “discovery page” will automatically move from `/actuator` to the root of the management context. For example, if the management context path is `/management` then the discovery page will be available from `/management`.

If the HAL Browser is on the classpath via its webjar (`org.webjars:hal-browser`), or via the `spring-data-rest-hal-browser` then an HTML “discovery page”, in the form of the HAL Browser, is also provided.

45.3 CORS support

Cross-origin resource sharing (CORS) is a W3C specification that allows you to specify in a flexible way what kind of cross domain requests are authorized. Actuator’s MVC endpoints can be configured to support such scenarios.

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

```
endpoints.cors.allowed-origins=http://example.com
endpoints.cors.allowed-methods=GET,POST
```

Tip

Check EndpointCorsProperties for a complete list of options.

45.4 Adding custom endpoints

If you add a `@Bean` of type `Endpoint` then it will automatically be exposed over JMX and HTTP (if there is an server available). An HTTP endpoints can be customized further by creating a bean of type `MvcEndpoint`. Your `MvcEndpoint` is not a `@Controller` but it can use `@RequestMapping` (and `@Managed*`) to expose resources.

Tip

If you are doing this as a library feature consider adding a configuration class to `/META-INF/spring.factories` under the key `org.springframework.boot.actuate.autoconfigure.EndpointWebMvcConfiguration`. If you do that then the endpoint will move to a child context with all the other MVC endpoints if your users ask for a separate management port or address. A configuration declared this way can be a `WebConfigurerAdapter` if it wants to add static resources (for instance) to the management endpoints.

45.5 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 unauthenticated connection in a secure application a simple ‘status’ message is returned, and for an authenticated connection additional details are also displayed (see Section 46.6, “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.

45.6 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
`CassandraHealthIndicator`	Checks that a Cassandra database is up.
`DiskSpaceHealthIndicator`	Checks for low disk space.
`DataSourceHealthIndicator`	Checks that a connection to `DataSource` can be obtained.
`ElasticsearchHealthIndicator`	Checks that an ElasticSearch cluster is up.
`JmsHealthIndicator`	Checks that a JMS broker is up.
`MailHealthIndicator`	Checks that a mail server is up.
`MongoHealthIndicator`	Checks that a Mongo database is up.
`RabbitHealthIndicator`	Checks that a Rabbit server is up.
`RedisHealthIndicator`	Checks that a Redis server is up.
`SolrHealthIndicator`	Checks that a Solr server is up.

Tip

It is possible to disable them all using the `management.health.defaults.enabled` property.

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 MyHealthIndicator implements HealthIndicator {

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

}
```

Note

The identifier for a given `HealthIndicator` is the name of the bean without the `HealthIndicator` suffix if it exists. In the example above, the health information will be available in an entry named `my`.

In addition to Spring Boot's predefined `Status` types, it is also possible for `Health` to return a custom `Status` that represents a new system state. In such cases a custom implementation of the `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`.

45.7 Application information

Application information exposes various information collected from all `InfoContributor` beans defined in your `ApplicationContext`. Spring Boot includes a number of auto-configured `InfoContributors` and you can also write your own.

Auto-configured InfoContributors

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

Name	Description
`EnvironmentInfoContributor`	Expose any key from the `Environment` under the `info` key.
`GitInfoContributor`	Expose git information if a `git.properties` file is available.
`BuildInfoContributor`	Expose build information if a `META-INF/boot/build.properties` file is available.

> **Tip**
>
> It is possible to disable them all using the `management.info.defaults.enabled` property.

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.encoding=UTF-8
info.app.java.source=1.8
info.app.java.target=1.8
```

> **Tip**
>
> Rather than hardcoding those values you could also expand info properties at build time.
>
> Assuming you are using Maven, you could rewrite the example above as follows:
>
> ```
> info.app.encoding=@project.build.sourceEncoding@
> info.app.java.source=@java.version@
> info.app.java.target=@java.version@
> ```

Git commit information

Another useful feature of the `info` endpoint is its ability to publish information about the state of your `git` source code repository when the project was built. If a `GitProperties` bean is available, the `git.branch`, `git.commit.id` and `git.commit.time` properties will be exposed.

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

If you want to display the full git information (i.e. the full content of `git.properties`), use the `management.info.git.mode` property:

```
management.info.git.mode=full
```

Build information

The `info` endpoint can also publish information about your build if a `BuildProperties` bean is available. This happens if a `META-INF/boot/build.properties` file is available in the classpath.

> **Tip**
>
> The Maven and Gradle plugins can both generate that file, see Generate build information for more details.

If additional properties are present they are not exposed unless configured explicitly:

```
management.info.build.mode=full
```

Writing custom InfoContributors

To provide custom application information you can register Spring beans that implement the `InfoContributor` interface.

The example below contributes an `example` entry with a single value:

```
import java.util.Collections;

import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class ExampleInfoContributor implements InfoContributor {

    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("example",
                Collections.singletonMap("key", "value"));
    }

}
```

If you hit the `info` endpoint you should see a response that contains the following additional entry:

```
{
    "example": {
        "key" : "value"
    }
}
```

46. Monitoring and management over HTTP

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`.

46.1 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 45.1, “Customizing endpoints” for details of how you can set `endpoints.enabled` to `false` then “opt-in” only specific endpoints.

46.2 Customizing the management endpoint paths

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.context-path` 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`).

You can also change the “id” of an endpoint (using `endpoints.{name}.id`) which then changes the default resource path for the MVC endpoint. Legal endpoint ids are composed only of alphanumeric characters (because they can be exposed in a number of places, including JMX object names, where special characters are forbidden). The MVC path can be changed separately by configuring `endpoints.{name}.path`, and there is no validation on those values (so you can use anything that is legal in a URL path). For example, to change the location of the `/health` endpoint to `/ping/me` you can set `endpoints.health.path=/ping/me`.

> **Tip**
>
> If you provide a custom `MvcEndpoint` remember to include a settable `path` property, and default it to `/{id}` if you want your code to behave like the standard MVC endpoints. (Take a look at the `HealthMvcEndpoint` to see how you might do that.) If your custom endpoint is an `Endpoint` (not an `MvcEndpoint`) then Spring Boot will take care of the path for you.

46.3 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.)

46.4 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
```

46.5 Disabling HTTP endpoints

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

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

46.6 HTTP health endpoint access restrictions

The information exposed by the health endpoint varies depending on whether or not it’s accessed anonymously, and whether or not the enclosing application is secure. By default, when accessed anonymously in a secure application, 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 enhanced, thereby allowing only authenticated users full access to the health endpoint in a secure application. To do so, set `endpoints.health.sensitive` to `true`. Here's a summary of behavior (with default `sensitive` flag value "false" indicated in bold):

`management.security.enabled`	`endpoints.health.sensitive`	Unauthenticated	Authenticated
false	**false**	Full content	Full content
false	true	Status only	Full content
true	**false**	Status only	Full content
true	true	No content	Full content

47. Monitoring and management over JMX

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.

47.1 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.unique-names` 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.unique-names=true
```

47.2 Disabling JMX endpoints

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

```
endpoints.jmx.enabled=false
```

47.3 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
```

48. Monitoring and management using a remote shell

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 the following dependency to your project:

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

Tip

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

Note

CRaSH requires to run with a JDK as it compiles commands on the fly. If a basic `help` command fails, you are probably running with a JRE.

48.1 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 ::  (v1.4.0.M2) 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.

48.2 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.command-path-patterns` property.

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

```
package commands

import org.crsh.cli.Command
import org.crsh.cli.Usage
import org.crsh.command.InvocationContext

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.

49. Metrics

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,
    "nonheap": 0,
    "nonheap.committed": 77568,
    "nonheap.init": 2496,
    "nonheap.used": 75826,
    "mem": 986624,
    "mem.free": 933858,
    "processors": 8,
    "threads": 15,
    "threads.daemon": 11,
    "threads.peak": 15,
    "threads.totalStarted": 42,
    "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 asterisks (`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.

49.1 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`, `thread.daemon`)
- Class load information (`classes`, `classes.loaded`, `classes.unloaded`)
- Garbage collection information (`gc.xxx.count`, `gc.xxx.time`)

49.2 DataSource metrics

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

- 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 data sources; you can add additional `DataSourcePoolMetadataProvider` beans if your favorite data source isn’t supported out of the box. See `DataSourcePoolMetadataProvidersConfiguration` for examples.

49.3 Cache metrics

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

- The current size of the cache (`cache.xxx.size`)
- Hit ratio (`cache.xxx.hit.ratio`)
- Miss ratio (`cache.xxx.miss.ratio`)

Note

Cache providers do not expose the hit/miss ratio in a consistent way. While some expose an **aggregated** value (i.e. the hit ratio since the last time the stats were cleared), others expose a

> **temporal** value (i.e. the hit ratio of the last second). Check your caching provider documentation for more details.

If two different cache managers happen to define the same cache, the name of the cache is prefixed by the name of the `CacheManager` bean.

It is possible to override part or all of those defaults by registering a bean with a customized version of `CachePublicMetrics`. By default, Spring Boot provides cache statistics for EhCache, Hazelcast, Infinispan, JCache and Guava. You can add additional `CacheStatisticsProvider` beans if your favorite caching library isn't supported out of the box. See `CacheStatisticsAutoConfiguration` for examples.

49.4 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.

49.5 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.

49.6 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`.

49.7 Special features with Java 8

The default implementation of `GaugeService` and `CounterService` provided by Spring Boot depends on the version of Java that you are using. With Java 8 (or better) the implementation switches to a high-performance version optimized for fast writes, backed by atomic in-memory buffers, rather than by the immutable but relatively expensive `Metric<?>` type (counters are approximately 5 times faster and gauges approximately twice as fast as the repository-based implementations). The Dropwizard metrics services (see below) are also very efficient even for Java 7 (they have backports of some of the Java 8 concurrency libraries), but they do not record timestamps for metric values. If performance of metric gathering is a concern then it is always advisable to use one of the high-performance options, and also to only read metrics infrequently, so that the writes are buffered locally and only read when needed.

> **Note**
>
> The old `MetricRepository` and its `InMemoryMetricRepository` implementation are not used by default if you are on Java 8 or if you are using Dropwizard metrics.

49.8 Metric writers, exporters and aggregation

Spring Boot provides a couple of implementations of a marker interface called `Exporter` which can be used to copy metric readings from the in-memory buffers to a place where they can be analyzed and displayed. Indeed, if you provide a `@Bean` that implements the `MetricWriter` interface (or `GaugeWriter` for simple use cases) and mark it `@ExportMetricWriter`, then it will automatically be hooked up to an `Exporter` and fed metric updates every 5 seconds (configured via `spring.metrics.export.delay-millis`). In addition, any `MetricReader` that you define and mark as `@ExportMetricReader` will have its values exported by the default exporter.

The default exporter is a `MetricCopyExporter` which tries to optimize itself by not copying values that haven’t changed since it was last called (the optimization can be switched off using a flag `spring.metrics.export.send-latest`). Note also that the Dropwizard `MetricRegistry` has no support for timestamps, so the optimization is not available if you are using Dropwizard metrics (all metrics will be copied on every tick).

The default values for the export trigger (`delay-millis`, `includes`, `excludes` and `send-latest`) can be set as `spring.metrics.export.*`. Individual values for specific `MetricWriters` can be set as `spring.metrics.export.triggers.<name>.*` where <name> is a bean name (or pattern for matching bean names).

> **Warning**
>
> The automatic export of metrics is disabled if you switch off the default `MetricRepository` (e.g. by using Dropwizard metrics). You can get back the same functionality be declaring a bean of your own of type `MetricReader` and declaring it to be `@ExportMetricReader`.

Example: Export to Redis

If you provide a `@Bean` of type `RedisMetricRepository` and mark it `@ExportMetricWriter` the metrics are exported to a Redis cache for aggregation. The `RedisMetricRepository` has two important parameters to configure it for this purpose: `prefix` and `key` (passed into its constructor). It is best to use a prefix that is unique to the application instance (e.g. using a random value and maybe the logical name of the application to make it possible to correlate with other instances of the same application). The “key” is used to keep a global index of all metric names, so it should be unique “globally”, whatever that means for your system (e.g. two instances of the same system could share a Redis cache if they have distinct keys).

Example:

```
@Bean
@ExportMetricWriter
MetricWriter metricWriter(MetricExportProperties export) {
 return new RedisMetricRepository(connectionFactory,
      export.getRedis().getPrefix(), export.getRedis().getKey());
}
```

application.properties.

```
spring.metrics.export.redis.prefix: metrics.mysystem.${spring.application.name:application}.
${random.value:0000}
spring.metrics.export.redis.key: keys.metrics.mysystem
```

The prefix is constructed with the application name and id at the end, so it can easily be used to identify a group of processes with the same logical name later.

Note

It’s important to set both the `key` and the `prefix`. The key is used for all repository operations, and can be shared by multiple repositories. If multiple repositories share a key (like in the case where you need to aggregate across them), then you normally have a read-only “master” repository that has a short, but identifiable, prefix (like “metrics.mysystem”), and many write-only repositories with prefixes that start with the master prefix (like `metrics.mysystem.*` in the example above). It is efficient to read all the keys from a “master” repository like that, but inefficient to read a subset with a longer prefix (e.g. using one of the writing repositories).

Tip

The example above uses `MetricExportProperties` to inject and extract the key and prefix. This is provided to you as a convenience by Spring Boot, configured with sensible defaults. There is nothing to stop you using your own values as long as they follow the recommendations.

Example: Export to Open TSDB

If you provide a `@Bean` of type `OpenTsdbGaugeWriter` and mark it `@ExportMetricWriter` metrics are exported to Open TSDB for aggregation. The `OpenTsdbGaugeWriter` has a `url` property that you need to set to the Open TSDB “/put” endpoint, e.g. `localhost:4242/api/put`). It also has a `namingStrategy` that you can customize or configure to make the metrics match the data structure you need on the server. By default it just passes through the metric name as an Open TSDB metric name, and adds the tags “domain” (with value “org.springframework.metrics”) and “process” (with the value equal to the object hash of the naming strategy). Thus, after running the application and generating some metrics you can inspect the metrics in the TSD UI (localhost:4242 by default).

Example:

```
curl localhost:4242/api/query?start=1h-ago&m=max:counter.status.200.root
[
 {
  "metric": "counter.status.200.root",
  "tags": {
   "domain": "org.springframework.metrics",
   "process": "b968a76"
  },
  "aggregateTags": [],
  "dps": {
   "1430492872": 2,
   "1430492875": 6
  }
 }
]
```

Example: Export to Statsd

To export metrics to Statsd, make sure first that you have added `com.timgroup:java-statsd-client` as a dependency of your project (Spring Boot provides a dependency management for it). Then add a `spring.metrics.export.statsd.host` value to your `application.properties` file. Connections will be opened to port `8125` unless a `spring.metrics.export.statsd.port` override is provided. You can use `spring.metrics.export.statsd.prefix` if you want a custom prefix.

Alternatively, you can provide a `@Bean` of type `StatsdMetricWriter` and mark it `@ExportMetricWriter`:

```
@Value("${spring.application.name:application}.${random.value:0000}")
private String prefix = "metrics";

@Bean
@ExportMetricWriter
MetricWriter metricWriter() {
 return new StatsdMetricWriter(prefix, "localhost", "8125");
}
```

Example: Export to JMX

If you provide a `@Bean` of type `JmxMetricWriter` marked `@ExportMetricWriter` the metrics are exported as MBeans to the local server (the `MBeanExporter` is provided by Spring Boot JMX auto-configuration as long as it is switched on). Metrics can then be inspected, graphed, alerted etc. using any tool that understands JMX (e.g. JConsole or JVisualVM).

Example:

```
@Bean
@ExportMetricWriter
MetricWriter metricWriter(MBeanExporter exporter) {
 return new JmxMetricWriter(exporter);
}
```

Each metric is exported as an individual MBean. The format for the `ObjectNames` is given by an `ObjectNamingStrategy` which can be injected into the `JmxMetricWriter` (the default breaks up the metric name and tags the first two period-separated sections in a way that should make the metrics group nicely in JVisualVM or JConsole).

49.9 Aggregating metrics from multiple sources

There is an `AggregateMetricReader` that you can use to consolidate metrics from different physical sources. Sources for the same logical metric just need to publish them with a period-separated prefix, and the reader will aggregate (by truncating the metric names, and dropping the prefix). Counters are summed and everything else (i.e. gauges) take their most recent value.

This is very useful if multiple application instances are feeding to a central (e.g. Redis) repository and you want to display the results. Particularly recommended in conjunction with a `MetricReaderPublicMetrics` for hooking up to the results to the “/metrics” endpoint.

Example:

```
@Autowired
private MetricExportProperties export;

@Bean
public PublicMetrics metricsAggregate() {
  return new MetricReaderPublicMetrics(aggregatesMetricReader());
}

private MetricReader globalMetricsForAggregation() {
  return new RedisMetricRepository(this.connectionFactory,
      this.export.getRedis().getAggregatePrefix(), this.export.getRedis().getKey());
}

private MetricReader aggregatesMetricReader() {
  AggregateMetricReader repository = new AggregateMetricReader(
      globalMetricsForAggregation());
  return repository;
}
```

Note

The example above uses `MetricExportProperties` to inject and extract the key and prefix. This is provided to you as a convenience by Spring Boot, and the defaults will be sensible. They are set up in `MetricExportAutoConfiguration`.

Note

The `MetricReaders` above are not `@Beans` and are not marked as `@ExportMetricReader` because they are just collecting and analyzing data from other repositories, and don’t want to export their values.

49.10 Dropwizard Metrics

A default `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. Users of the Dropwizard ‘Metrics’ library will find that Spring Boot metrics are automatically published to `com.codahale.metrics.MetricRegistry`. Metrics from the `MetricRegistry` are also automatically exposed via the `/metrics` endpoint

When Dropwizard metrics are in use, the default `CounterService` and `GaugeService` are replaced with a `DropwizardMetricServices`, which is a wrapper around the `MetricRegistry` (so you can `@Autowired` one of those services and use it as normal). You can also create “special” Dropwizard metrics by prefixing your metric names with the appropriate type (i.e. `timer.*`, `histogram.*` for gauges, and `meter.*` for counters).

49.11 Message channel integration

If a `MessageChannel` bean called `metricsChannel` exists, then a `MetricWriter` will be created that writes metrics to that channel. The writer is automatically hooked up to an exporter (as for all writers), so all metric values will appear on the channel, and additional analysis or actions can be taken by subscribers (it’s up to you to provide the channel and any subscribers you need).

50. Auditing

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. To customize published security events you can provide your own implementations of `AbstractAuthenticationAuditListener` and `AbstractAuthorizationAuditListener`.

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`).

51. Tracing

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,
    ...
    }]
```

51.1 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.

52. Process monitoring

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.

52.1 Extend configuration

In `META-INF/spring.factories` file you can activate the listener(s) that writes a PID file. Example:

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

52.2 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.

53. What to read next

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 ‘deployment options’ or jump ahead for some in-depth information about Spring Boot’s *build tool plugins*.

Part VI. Deploying Spring Boot applications

Spring Boot’s flexible packaging options provide a great deal of choice when it comes to deploying your application. You can easily deploy Spring Boot applications to a variety of cloud platforms, to a container images (such as Docker) or to virtual/real machines.

This section covers some of the more common deployment scenarios.

54. 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 web server, 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 development and production environments.

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

In this section we'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.

54.1 Cloud Foundry

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

Once you'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 `http://acloudyspringtime.cfapps.io/`.

Binding to services

By default, metadata about the running application as well as service connection information is exposed to the application as environment variables (for example: `$VCAP_SERVICES`). This architecture decision is due to Cloud Foundry's polyglot (any language and platform can be supported as a buildpack) nature; process-scoped environment variables are language agnostic.

Environment variables 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` Javadoc 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.

54.2 Heroku

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

You must configure your application to listen on the correct port. 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 1.8. 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 1.8... done
-----> Installing Maven 3.3.1... done
-----> Installing settings.xml... done
-----> Executing: mvn -B -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.

54.3 OpenShift

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.

54.4 Boxfuse and Amazon Web Services

Boxfuse works by turning your Spring Boot executable jar or war into a minimal VM image that can be deployed unchanged either on VirtualBox or on AWS. Boxfuse comes with deep integration for Spring Boot and will use the information from your Spring Boot configuration file to automatically configure ports and health check URLs. Boxfuse leverages this information both for the images it produces as well as for all the resources it provisions (instances, security groups, elastic load balancers, etc).

Once you have created a Boxfuse account, connected it to your AWS account, and installed the latest version of the Boxfuse Client, you can deploy your Spring Boot application to AWS as follows (ensure the application has been built by Maven or Gradle first using, for example, `mvn clean package`):

```
$ boxfuse run myapp-1.0.jar -env=prod
```

See the `boxfuse run` documentation for more options. If there is a boxfuse.com/docs/commandline/#configuration [`boxfuse.conf`] file present in the current directory, it will be consulted.

Tip

By default Boxfuse will activate a Spring profile named `boxfuse` on startup and if your executable jar or war contains an boxfuse.com/docs/payloads/springboot.html#configuration [`application-boxfuse.properties`] file, Boxfuse will base its configuration based on the properties it contains.

At this point `boxfuse` will create an image for your application, upload it, and then configure and start the necessary resources on AWS:

```
Fusing Image for myapp-1.0.jar ...
Image fused in 00:06.838s (53937 K) -> axelfontaine/myapp:1.0
Creating axelfontaine/myapp ...
Pushing axelfontaine/myapp:1.0 ...
Verifying axelfontaine/myapp:1.0 ...
Creating Elastic IP ...
Mapping myapp-axelfontaine.boxfuse.io to 52.28.233.167 ...
Waiting for AWS to create an AMI for axelfontaine/myapp:1.0 in eu-central-1 (this may take up to 50
 seconds) ...
AMI created in 00:23.557s -> ami-d23f38cf
Creating security group boxfuse-sg_axelfontaine/myapp:1.0 ...
Launching t2.micro instance of axelfontaine/myapp:1.0 (ami-d23f38cf) in eu-central-1 ...
Instance launched in 00:30.306s -> i-92ef9f53
Waiting for AWS to boot Instance i-92ef9f53 and Payload to start at http://52.28.235.61/ ...
Payload started in 00:29.266s -> http://52.28.235.61/
Remapping Elastic IP 52.28.233.167 to i-92ef9f53 ...
Waiting 15s for AWS to complete Elastic IP Zero Downtime transition ...
Deployment completed successfully. axelfontaine/myapp:1.0 is up and running at http://myapp-
axelfontaine.boxfuse.io/
```

Your application should now be up and running on AWS.

There’s a blog on deploying Spring Boot apps on EC2 as well as documentation for the Boxfuse Spring Boot integration on their website that will get you started with a Maven build to run the app.

54.5 Google App Engine

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.

55. Installing Spring Boot applications

In additional to running Spring Boot applications using `java -jar` it is also possible to make fully executable applications for Unix systems (Linux, OSX, FreeBSD etc). This makes it very easy to install and manage Spring Boot applications in common production environments. As long as you are generating ‘fully executable’ jars from your build, and you are not using a custom `embeddedLaunchScript`, the following techniques can be used.

To create a ‘fully executable’ jar with Maven use the following plugin configuration:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <executable>true</executable>
    </configuration>
</plugin>
```

With Gradle, the equivalent configuration would be:

```
apply plugin: 'spring-boot'

springBoot {
    executable = true
}
```

Note

Fully executable jars work by embedding an extra script at the front of the file. Not all tools currently accept this format so you may not always be able to use this technique.

Note

When a fully executable jar is run, it uses the jar’s directory as the working directory.

55.1 Unix/Linux services

Spring Boot application can be easily started as Unix/Linux services using either `init.d` or `systemd`.

Installation as an init.d service (System V)

The default executable script that can be embedded into Spring Boot jars will act as an `init.d` script when it is symlinked to `/etc/init.d`. The standard `start`, `stop`, `restart` and `status` commands can be used. The script supports the following features:

- Starts the services as the user that owns the jar file
- Tracks application’s PID using `/var/run/<appname>/<appname>.pid`
- Writes console logs to `/var/log/<appname>.log`

Assuming that you have a Spring Boot application installed in `/var/myapp`, to install a Spring Boot application as an `init.d` service simply create a symlink:

```
$ sudo ln -s /var/myapp/myapp.jar /etc/init.d/myapp
```

Once installed, you can start and stop the service in the usual way. You can also flag the application to start automatically using your standard operating system tools. For example, if you use Debian:

```
$ update-rc.d myapp defaults <priority>
```

Securing an init.d service

> **Note**
>
> The following is a set of guidelines on how to secure a Spring Boot application that's being run as an init.d service. It is not intended to be an exhaustive list of everything that should be done to harden an application and the environment in which it runs.

When executed as root, as is the case when root is being used to start an init.d service, the default executable script will run the application as the user which owns the jar file. You should never run a Spring Boot application as `root` so your application's jar file should never be owned by root. Instead, create a specific user to run your application and use `chown` to make it the owner of the jar file. For example:

```
$ chown bootapp:bootapp your-app.jar
```

In this case, the default executable script will run the application as the `bootapp` user.

> **Tip**
>
> To reduce the chances of the application's user account being compromised, you should consider preventing it from using a login shell. Set the account's shell to `/usr/sbin/nologin`, for example.

You should also take steps to prevent the modification of your application's jar file. Firstly, configure its permissions so that it cannot be written and can only be read or executed by its owner:

```
$ chmod 500 your-app.jar
```

Secondly, you should also take steps to limit the damage if your application or the account that's running it is compromised. If an attacker does gain access, they could make the jar file writable and change its contents. One way to protect against this is to make it immutable using `chattr`:

```
$ sudo chattr +i your-app.jar
```

This will prevent any user, including root, from modifying the jar.

If root is used to control the application's service and you use a `.conf` file to customize its startup, the `.conf` file will be read and evaluated by the root user. It should be secured accordingly. Use `chmod` so that the file can only be read by the owner and use `chown` to make root the owner:

```
$ chmod 400 your-app.conf
$ sudo chown root:root your-app.conf
```

Installation as a systemd service

Systemd is the successor of the System V init system, and is now being used by many modern Linux distributions. Although you can continue to use `init.d` scripts with `systemd`, it is also possible to launch Spring Boot applications using `systemd` 'service' scripts.

Assuming that you have a Spring Boot application installed in `/var/myapp`, to install a Spring Boot application as a `systemd` service create a script named `myapp.service` using the following example and place it in `/etc/systemd/system` directory:

```
[Unit]
Description=myapp
After=syslog.target

[Service]
User=myapp
ExecStart=/var/myapp/myapp.jar
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```

Tip

Remember to change the `Description`, `User` and `ExecStart` fields for your application.

Note that unlike when running as an `init.d` service, user that runs the application, PID file and console log file behave differently under `systemd` and must be configured using appropriate fields in 'service' script. Consult the service unit configuration man page for more details.

To flag the application to start automatically on system boot use the following command:

```
$ systemctl enable myapp.service
```

Refer to `man systemctl` for more details.

Customizing the startup script

The script accepts the following parameters as environment variables, so you can change the default behavior in a script or on the command line:

Variable	Description
`MODE`	The "mode" of operation. The default depends on the way the jar was built, but will usually be `auto` *(meaning it tries to guess if it is an init script by checking if it is a symlink in a directory called `init.d`)*. You can explicitly set it to `service` so that the `stop|start|status|restart` commands work, or to `run` if you just want to run the script in the foreground.
`USE_START_STOP_DAEMON`	If the `start-stop-daemon` command, when it's available, should be used to control the process. Defaults to `true`.
`PID_FOLDER`	The root name of the pid folder (`/var/run` by default).
`LOG_FOLDER`	The name of the folder to put log files in (`/var/log` by default).
`CONF_FOLDER`	The name of the folder to read .conf files from (same folder as jar-file by default).
`LOG_FILENAME`	The name of the log file in the `LOG_FOLDER` (`<appname>.log` by default).
`APP_NAME`	The name of the app. If the jar is run from a symlink the script guesses the app name, but if it is not a symlink, or you want to explicitly set the app name this can be useful.
`RUN_ARGS`	The arguments to pass to the program (the Spring Boot app).

Variable	Description
`JAVA_HOME`	The location of the `java` executable is discovered by using the `PATH` by default, but you can set it explicitly if there is an executable file at `$JAVA_HOME/bin/java`.
`JAVA_OPTS`	Options that are passed to the JVM when it is launched.
`JARFILE`	The explicit location of the jar file, in case the script is being used to launch a jar that it is not actually embedded in.
`DEBUG`	if not empty will set the `-x` flag on the shell process, making it easy to see the logic in the script.

Note

The `PID_FOLDER`, `LOG_FOLDER` and `LOG_FILENAME` variables are only valid for an `init.d` service. With `systemd` the equivalent customizations are made using 'service' script. Check the service unit configuration man page for more details.

In addition, the following properties can be changed when the script is written by using the `embeddedLaunchScriptProperties` option of the Spring Boot Maven or Gradle plugins.

Name	Description
`mode`	The script mode. Defaults to `auto`.
`initInfoProvides`	The `Provides` section of "INIT INFO". Defaults to `spring-boot-application` for Gradle and to `${project.artifactId}` for Maven.
`initInfoShortDescription`	The `Short-Description` section of "INIT INFO". Defaults to `Spring Boot Application` for Gradle and to `${project.name}` for Maven.
`initInfoDescription`	The `Description` section of "INIT INFO". Defaults to `Spring Boot Application` for Gradle and to `${project.description}` (falling back to `${project.name}`) for Maven.
`initInfoChkconfig`	The `chkconfig` section of "INIT INFO". Defaults to `2345 99 01`.
`logFolder`	The default value for `LOG_FOLDER`. Only valid for an `init.d` service.
`pidFolder`	The default value for `PID_FOLDER`. Only valid for an `init.d` service.
`useStartStopDaemon`	If the `start-stop-daemon` command, when it's available, should be used to control the process. Defaults to `true`.

Customizing the startup script with a conf file

With the exception of `JARFILE` and `APP_NAME`, the above settings can be configured using a `.conf` file,

```
JAVA_OPTS=-Xmx1024M
LOG_FOLDER=/custom/log/folder
```

The file is expected next to the jar file and have the same name but suffixed with `.conf` rather than `.jar`. For example, a jar named `/var/myapp/myapp.jar` will use the configuration file named `/var/myapp/myapp.conf` if it exists. You can also use the `CONF_FOLDER` property to customize the location of that file.

To learn about securing this file appropriately, please refer to the guidelines for securing an init.d service.

56. Microsoft Windows services

Spring Boot application can be started as Windows service using `winsw`.

A sample maintained separately to the core of Spring Boot describes step-by-step how you can create a Windows service for your Spring Boot application.

57. What to read next

Check out the Cloud Foundry, Heroku, OpenShift and Boxfuse web sites for more information about the kinds of features that a PaaS can offer. These are just four of the most popular Java PaaS providers, since Spring Boot is so amenable to cloud-based deployment you’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.

58. Installing the CLI

The Spring Boot CLI can be installed manually; using SDKMAN! (the SDK Manager) or using Homebrew or MacPorts if you are an OSX user. See *Section 10.2, “Installing the Spring Boot CLI”* in the “Getting started” section for comprehensive installation instructions.

59. Using the CLI

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 v1.4.0.M2
```

59.1 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 dependency management

By default, the CLI uses the dependency management declared in `spring-boot-dependencies` when resolving `@Grab` dependencies. Additional dependency management, that will override the default dependency management, can be configured using the `@DependencyManagementBom` annotation. The annotation’s value should specify the coordinates (`groupId:artifactId:version`) of one or more Maven BOMs.

For example, the following declaration:

```
@DependencyManagementBom("com.example.custom-bom:1.0.0")
```

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

When multiple BOMs are specified they are applied in the order that they’re declared. For example:

```
@DependencyManagementBom(["com.example.custom-bom:1.0.0",
        "com.example.another-bom:1.0.0"])
```

indicates that dependency management in `another-bom` will override the dependency management in `custom-bom`.

You can use `@DependencyManagementBom` anywhere that you can use `@Grab`, however, to ensure consistent ordering of the dependency management, you can only use `@DependencyManagementBom` at most once in your application. A useful source of dependency management (that is a superset of Spring Boot’s dependency management) is the Spring IO Platform, e.g. `@DependencyManagementBom('io.spring.platform:platform-bom:1.1.2.RELEASE')`.

59.2 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.

59.3 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
```

59.4 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.

59.5 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'
```

59.6 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 (v1.4.0.M2)
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 v1.4.0.M2
```

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.

59.7 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
```

60. Developing application with the Groovy beans DSL

Spring Framework 4.0 has native support for a `beans{}` “DSL” (borrowed from Grails), and you can embed bean definitions in your Groovy application scripts 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.

61. Configuring the CLI with settings.xml

The Spring Boot CLI uses Aether, Maven's dependency resolution engine, to resolve dependencies. The CLI makes use of the Maven configuration found in `~/.m2/settings.xml` to configure Aether. The following configuration settings are honored by the CLI:

- Offline
- Mirrors
- Servers
- Proxies
- Profiles
  - Activation
  - Repositories
- Active profiles

Please refer to Maven's settings documentation for further information.

62. What to read next

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.

63. Spring Boot Maven plugin

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.

63.1 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>1.4.0.M2</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>
```

63.2 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 80.1, “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.

64. Spring Boot Gradle plugin

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 use the dependency management provided by `spring-boot-dependencies`.

64.1 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:1.4.0.M2")
    }
}
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"
    }
    // ...
}
```

64.2 Gradle dependency management

The `spring-boot` plugin automatically applies the Dependency Management Plugin and configures in to import the `spring-boot-starter-parent` bom. This provides a similar dependency management experience to the one that is enjoyed by Maven users. For example, it allows you to omit version numbers when declaring dependencies that are managed in the bom. 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 version of the `spring-boot-starter-parent` bom that is imported (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 dependency management 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")
}
```

To learn more about the capabilities of the Dependency Management Plugin, please refer to its documentation.

64.3 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.

Tip

Check Section 64.6, "Repackage configuration" for a full list of configuration options.

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 80.1, "Create a deployable war file"" section for more details on how to create a deployable war file.

64.4 Running a project in-place

To run a project in place without building a jar first you can use the "bootRun" task:

```
$ gradle bootRun
```

If `devtools` has been added to your project it will automatically monitor your application for changes. Alternatively, you can also run the application so that your static classpath resources (i.e. in `src/main/resources` by default) are reloadable in the live application, which can be helpful at development time.

```
bootRun {
    addResources = true
}
```

Making static classpath resources reloadable means that `bootRun` does not use the output of the `processResources` task, i.e., when invoked using `bootRun`, your application will use the resources in their unprocessed form.

64.5 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
}
```

64.6 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, and you have applied the application plugin, the `mainClassName` project property will be used. If the application plugin has not been applied or no `mainClassName` has been specified, 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). If you have applied the application plugin, the main class can also be specified via its "run" task (`main` property) and/or its "startScripts" task (`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

Name	Description
	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 which is used to populate the nested lib directory (without specifying this you get all compile and runtime dependencies).
`executable`	Boolean flag to indicate if jar files are fully executable on Unix like operating systems. Defaults to `false`.
`embeddedLaunchScript`	The embedded launch script to prepend to the front of the jar if it is fully executable. If not specified the 'Spring Boot' default script will be used.
`embeddedLaunchScriptProperties`	Additional properties that to be expanded in the launch script. The default script supports a `mode` property which can contain the values `auto`, `service` or `run`.
`excludeDevtools`	Boolean flag to indicate if the devtools jar should be excluded from the repackaged archives. Defaults to `false`.

64.7 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).
`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.

64.8 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.

64.9 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 "1.4.0.M2"
                    }
                }
            }
        }
    }
}
```

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 "1.4.0.M2"
                                type "pom"
                                scope "import"
                            }
                        }
                    }
                }
            }
        }
    }
}
```

65. Spring Boot AntLib module

The Spring Boot AntLib module provides basic Spring Boot support for Apache Ant. You can use the module to create executable jars. To use the module you need to declare an additional `spring-boot` namespace in your `build.xml`:

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

You'll need to remember to start Ant using the `-lib` option, for example:

```
$ ant -lib <folder containing spring-boot-antlib-1.4.0.M2.jar>
```

Tip

The "Using Spring Boot" section includes a more complete example of using Apache Ant with `spring-boot-antlib`

65.1 Spring Boot Ant tasks

Once the `spring-boot-antlib` namespace has been declared, the following additional tasks are available.

spring-boot:exejar

The `exejar` task can be used to creates a Spring Boot executable jar. The following attributes are supported by the task:

Attribute	Description	Required
`destfile`	The destination jar file to create	Yes
`classes`	The root directory of Java class files	Yes
`start-class`	The main application class to run	No *(default is first class found declaring a `main` method)*

The following nested elements can be used with the task:

Element	Description
`resources`	One or more Resource Collections describing a set of Resources that should be added to the content of the created jar file.
`lib`	One or more Resource Collections that should be added to the set of jar libraries that make up the runtime dependency classpath of the application.

Examples

Specify start-class.

```
<spring-boot:exejar destfile="target/my-application.jar"
        classes="target/classes" start-class="com.foo.MyApplication">
    <resources>
        <fileset dir="src/main/resources" />
    </resources>
    <lib>
        <fileset dir="lib" />
    </lib>
</spring-boot:exejar>
```

Detect start-class.

```
<exejar destfile="target/my-application.jar" classes="target/classes">
    <lib>
        <fileset dir="lib" />
    </lib>
</exejar>
```

65.2 spring-boot:findmainclass

The `findmainclass` task is used internally by `exejar` to locate a class declaring a `main`. You can also use this task directly in your build if needed. The following attributes are supported

Attribute	Description	Required
`classesroot`	The root directory of Java class files	Yes *(unless `mainclass` is specified)*
`mainclass`	Can be used to short-circuit the `main` class search	No
`property`	The Ant property that should be set with the result	No *(result will be logged if unspecified)*

Examples

Find and log.

```
<findmainclass classesroot="target/classes" />
```

Find and set.

```
<findmainclass classesroot="target/classes" property="main-class" />
```

Override and set.

```
<findmainclass mainclass="com.foo.MainClass" property="main-class" />
```

66. Supporting other build systems

If you want to use a build tool other than Maven, Gradle or Ant, 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.

66.1 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.

66.2 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`.

66.3 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.

66.4 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));
            }
        });
```

67. What to read next

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.

68. Spring Boot application

68.1 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 `ConditionEvaluationReport` 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 auto-configuration 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 `RelaxedPropertyResolver` 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 `RelaxedPropertyResolver` 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 placeholders resolved from the `Environment`.

68.2 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 23.4, “Application events and listeners”* in the ‘Spring Boot features’ section for a complete list.

68.3 Build an ApplicationContext hierarchy (adding a parent or root context)

You can use the `ApplicationBuilder` class to create parent/child `ApplicationContext` hierarchies. See *Section 23.3, “Fluent builder API”* in the ‘Spring Boot features’ section for more information.

68.4 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 `setWebEnvironment(false)` on your `SpringApplication` instance, or set the `applicationContextClass` property (through the Java API or with external properties). Application code that you want to run as your business logic can be implemented as a `CommandLineRunner` and dropped into the context as a `@Bean` definition.

69. Properties & configuration

69.1 Automatically expand properties at build time

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

Automatic property expansion using Maven

You can automatically expand 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.

```
app.encoding=@project.build.sourceEncoding@
app.java.version=@java.version@
```

> **Tip**
>
> The `spring-boot:run` can add `src/main/resources` directly to the classpath (for hot reloading purposes) if you enable the `addResources` flag. This circumvents the resource filtering and this feature. You can use the `exec:java` goal instead or customize the plugin's configuration, see the plugin usage page for more details.

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>2.7</version>
    <configuration>
        <delimiters>
            <delimiter>@</delimiter>
        </delimiters>
        <useDefaultDelimiters>false</useDefaultDelimiters>
    </configuration>
</plugin>
```

> **Note**
>
> The `useDefaultDelimiters` property is important if you are using standard Spring placeholders in your configuration (e.g. `${foo}`). These may be expanded by the build if that property is not set to `false`.

Automatic property expansion using Gradle

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

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

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

```
app.name=${name}
app.description=${description}
```

Note

Gradle's `expand` method uses Groovy's `SimpleTemplateEngine` which transforms `${..}` tokens. The `${..}` style conflicts with Spring's own property placeholder mechanism. To use Spring property placeholders together with automatic expansion the Spring property placeholders need to be escaped like `\${..}`.

69.2 Externalize the configuration of SpringApplication

A `SpringApplication` has bean properties (mainly setters) so you can use its Java API as you create the application to modify its behavior. 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.banner-mode=off
```

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.

Properties defined in external configuration overrides the values specified via the Java API with the notable exception of the sources used to create the `ApplicationContext`. Let's consider this application

```
new SpringApplicationBuilder()
    .bannerMode(Banner.Mode.OFF)
    .sources(demo.MyApp.class)
    .run(args);
```

used with the following configuration:

```
spring.main.sources=com.acme.Config,com.acme.ExtraConfig
spring.main.banner-mode=console
```

The actual application will *now* show the banner (as overridden by configuration) and use three sources for the `ApplicationContext` (in that order): `demo.MyApp`, `com.acme.Config`, `com.acme.ExtraConfig`.

69.3 Change the location of external properties of an application

By default properties from different sources are added to the Spring `Environment` in a defined order (see *Chapter 24, 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 lower priority than any added using the default locations (e.g. `application.properties`), 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.

Spring Boot logs the configuration files that are loaded at `DEBUG` level and the candidates it has not found at `TRACE` level.

See `ConfigFileApplicationListener` for more detail.

69.4 Use ‘short’ command line arguments

Some people like to use (for example) `--port=9000` instead of `--server.port=9000` to set configuration properties on the command line. You can 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.

69.5 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 24.6, “Using YAML instead of Properties”* in the ‘Spring Boot features’ section for more information about YAML.

69.6 Set the active Spring profiles

The Spring `Environment` has an API for this, but normally you would set a System property (`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 25, Profiles* in the ‘Spring Boot features’ section for more information.

69.7 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.

69.8 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 `RelaxedPropertyResolver`.

70. Embedded servlet containers

70.1 Add a Servlet, Filter or Listener to an application

There are two ways to add `Servlet`, `Filter`, `ServletContextListener` and the other listeners supported by the Servlet spec to your application. You can either provide Spring beans for them, or enable scanning for Servlet components.

Add a Servlet, Filter or Listener using a Spring bean

To add a `Servlet`, `Filter`, or Servlet `*Listener` provide a `@Bean` definition for it. This can be very useful when you want to inject configuration or dependencies. However, you must 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.

> **Note**
>
> If no `dispatcherType` is specified on a filter registration, it will match `FORWARD`,`INCLUDE` and `REQUEST`. If async has been enabled, it will match `ASYNC` as well.
>
> If you are migrating a filter that has no `dispatcher` element in `web.xml` you will need to specify a `dispatcherType` yourself:
>
> ```
> @Bean
> public FilterRegistrationBean myFilterRegistration() {
>     FilterRegistrationBean registration = new FilterRegistrationBean();
>     registration.setDispatcherTypes(DispatcherType.REQUEST);
>     ....
>
>     return registration;
> }
> ```

Disable registration of a Servlet or Filter

As described above any `Servlet` or `Filter` beans will be registered with the servlet container automatically. To disable registration of a particular `Filter` or `Servlet` bean create a registration bean for it and mark it as disabled. For example:

```
@Bean
public FilterRegistrationBean registration(MyFilter filter) {
    FilterRegistrationBean registration = new FilterRegistrationBean(filter);
    registration.setEnabled(false);
    return registration;
}
```

Add Servlets, Filters, and Listeners using classpath scanning

`@WebServlet`, `@WebFilter`, and `@WebListener` annotated classes can be automatically registered with an embedded servlet container by annotating a `@Configuration` class with

`@ServletComponentScan` and specifying the package(s) containing the components that you want to register. By default, `@ServletComponentScan` will scan from the package of the annotated class.

70.2 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.

70.3 Use a random unassigned HTTP port

To scan for a free port (using OS natives to prevent clashes) use `server.port=0`.

70.4 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.

Tests that use `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)` can also inject the actual port into a field using the `@LocalServerPort` annotation. For example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class MyWebIntegrationTests {

    @Autowired
    EmbeddedWebApplicationContext server;

    @LocalServerPort
    int port;

    // ...

}
```

Note

`@LocalServerPort` is a meta-annotation for `@Value("${local.server.port}")`. Don’t try to inject the port in a regular application. As we just saw, the value is only set once the container has initialized; contrary to a test, application code callbacks are processed early (i.e. before the value is actually available).

70.5 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.

Using configuration like the example above means the application will no longer support plain HTTP connector at port 8080. Spring Boot doesn't support the configuration of both an HTTP connector and an HTTPS connector via `application.properties`. If you want to have both then you'll need to configure one of them programmatically. It's recommended to use `application.properties` to configure HTTPS as the HTTP connector is the easier of the two to configure programmatically. See the `spring-boot-sample-tomcat-multi-connectors` sample project for an example.

70.6 Configure Access Logging

Access logs can be configured for Tomcat and Undertow via their respective namespaces.

For instance, the following logs access on Tomcat with a custom pattern.

```
server.tomcat.basedir=my-tomcat
server.tomcat.accesslog.enabled=true
server.tomcat.accesslog.pattern=%t %a "%r" %s (%D ms)
```

Note

The default location for logs is a `logs` directory relative to the tomcat base dir and said directory is a temp directory by default so you may want to fix Tomcat's base directory or use an absolute path for the logs. In the example above, the logs will be available in `my-tomcat/logs` relative to the working directory of the application.

Access logging for undertow can be configured in a similar fashion

```
server.undertow.accesslog.enabled=true
server.undertow.accesslog.pattern=%t %a "%r" %s (%D ms)
```

Logs are stored in a `logs` directory relative to the working directory of the application. This can be customized via `server.undertow.accesslog.directory`.

70.7 Use behind a front-end proxy server

Your application might need to send `302` redirects or render content with absolute links back to itself. When running behind a proxy, the caller wants a link to the proxy, and not to the physical address of the machine hosting your app. Typically such situations are handled via a contract with the proxy, which will add headers to tell the back end how to construct links to itself.

If the proxy adds conventional `X-Forwarded-For` and `X-Forwarded-Proto` headers (most do this out of the box) the absolute links should be rendered correctly as long as `server.use-forward-headers` is set to `true` in your `application.properties`.

Note

If your application is running in Cloud Foundry or Heroku the `server.use-forward-headers` property will default to `true` if not specified. In all other instances it defaults to `false`.

Customize Tomcat’s proxy configuration

If you are using Tomcat you can additionally configure the names of the headers used to carry “forwarded” information:

```
server.tomcat.remote-ip-header=x-your-remote-ip-header
server.tomcat.protocol-header=x-your-protocol-header
```

Tomcat is also configured with a default regular expression that matches internal proxies that are to be trusted. By default, IP addresses in `10/8`, `192.168/16`, `169.254/16` and `127/8` are trusted. You can customize the valve’s configuration by adding an entry to `application.properties`, 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}`.

> **Note**
>
> You can trust all proxies by setting the `internal-proxies` to empty (but don’t do this in production).

You can take complete control of the configuration of Tomcat’s `RemoteIpValve` by switching the automatic one off (i.e. set `server.use-forward-headers=false`) and adding a new valve instance in a `TomcatEmbeddedServletContainerFactory` bean.

70.8 Configure Tomcat

Generally you can follow the advice from *Section 69.8, “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`.

70.9 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);
    }
}
```

70.10 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:1.4.0.M2")
    compile("org.springframework.boot:spring-boot-starter-jetty:1.4.0.M2")
    // ...
}
```

70.11 Configure Jetty

Generally you can follow the advice from *Section 69.8, “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`.

70.12 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:1.4.0.M2")
    compile("org.springframework.boot:spring-boot-starter-undertow:1.4.0.M2")
    // ...
}
```

70.13 Configure Undertow

Generally you can follow the advice from *Section 69.8, “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`.

70.14 Enable Multiple Listeners with Undertow

Add an `UndertowBuilderCustomizer` to the `UndertowEmbeddedServletContainerFactory` and add a listener to the `Builder`:

```
@Bean
public UndertowEmbeddedServletContainerFactory embeddedServletContainerFactory() {
    UndertowEmbeddedServletContainerFactory factory = new UndertowEmbeddedServletContainerFactory();
    factory.addBuilderCustomizers(new UndertowBuilderCustomizer() {

        @Override
        public void customize(Builder builder) {
            builder.addHttpListener(8080, "0.0.0.0");
        }

    });
    return factory;
}
```

70.15 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 .

Use Tomcat 7 with Maven

If you are using the starter poms and parent you can just change the Tomcat version property, e.g. for a simple webapp or service:

```
<properties>
    <tomcat.version>7.0.59</tomcat.version>
</properties>
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    ...
</dependencies>
```

Use Tomcat 7 with Gradle

You can change the Tomcat version by setting the `tomcat.version` property:

```
ext['tomcat.version'] = '7.0.59'
dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
}
```

70.16 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. You will also need to exclude Jetty's WebSocket-related dependencies.

Use Jetty 8 with Maven

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>
</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>
```

Use Jetty 8 with Gradle

You can set the `jetty.version` property and exclude the WebSocket dependency, e.g. for a simple webapp or service:

```
ext['jetty.version'] = '8.1.15.v20140411'
dependencies {
    compile ('org.springframework.boot:spring-boot-starter-web') {
        exclude group: 'org.springframework.boot', module: 'spring-boot-starter-tomcat'
    }
    compile ('org.springframework.boot:spring-boot-starter-jetty') {
        exclude group: 'org.eclipse.jetty.websocket'
    }
}
```

70.17 Create WebSocket endpoints using @ServerEndpoint

If you want to use `@ServerEndpoint` in a Spring Boot application that used an embedded container, you must declare a single `ServerEndpointExporter` `@Bean`:

```
@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.

70.18 Enable HTTP response compression

HTTP response compression is supported by Jetty, Tomcat, and Undertow. It can be enabled via `application.properties`:

```
server.compression.enabled=true
```

By default, responses must be at least 2048 bytes in length for compression to be performed. This can be configured using the `server.compression.min-response-size` property.

By default, responses will only be compressed if their content type is one of the following:

- `text/html`
- `text/xml`
- `text/plain`
- `text/css`

This can be configured using the `server.compression.mime-types` property.

71. Spring MVC

71.1 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.

71.2 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).

71.3 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 six enums in Jackson which map onto properties in the environment:

Jackson enum	Environment property
`com.fasterxml.jackson.databind.DeserializationFeature`	`spring.jackson.deserialization.<feature_name>=tru false`
`com.fasterxml.jackson.core.JsonGenerator.Feature`	`spring.jackson.generator.<feature_name>=true| false`
`com.fasterxml.jackson.databind.MapperFeature`	`spring.jackson.mapper.<feature_name>=true| false`
`com.fasterxml.jackson.core.JsonParser.Feature`	`spring.jackson.parser.<feature_name>=true| false`
`com.fasterxml.jackson.databind.SerializationFeature`	`spring.jackson.serialization.<feature_name>=true| false`
`com.fasterxml.jackson.annotation.JsonInclude.Include`	`spring.jackson.serialization-inclusion=always|non_null| non_absent|non_default|non_empty`

For example, to enable pretty print, set `spring.jackson.serialization.indent_output=true`. Note that, thanks to the use of relaxed binding, the case of `indent_output` 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 71.4, “Customize the @ResponseBody rendering”* section and the `WebMvcAutoConfiguration` source code for more details.

71.4 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.

71.5 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.

71.6 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).

71.7 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.

71.8 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.mvc.view.prefix` and `spring.mvc.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 `GroovyMarkupViewResolver` with id ‘groovyMarkupViewResolver’. It looks for resources in a loader path by surrounding the view name with a prefix and suffix (externalized to `spring.groovy.template.prefix` and `spring.groovy.template.suffix`, defaults ‘classpath:/templates/’ and ‘.tpl’ respectively). It can be overridden 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`

71.9 Velocity

By default, Spring Boot configures a `VelocityViewResolver`. If you need a `VelocityLayoutViewResolver` instead, you can easily configure your own by creating a bean with name `velocityViewResolver`. You can also inject the `VelocityProperties` instance to apply the base defaults to your custom view resolver.

The following example replaces the auto-configured velocity view resolver with a `VelocityLayoutViewResolver` defining a customized `layoutUrl` and all settings that would have been applied from the auto-configuration:

```
@Bean(name = "velocityViewResolver")
public VelocityLayoutViewResolver velocityViewResolver(VelocityProperties properties) {
    VelocityLayoutViewResolver resolver = new VelocityLayoutViewResolver();
    properties.applyToViewResolver(resolver);
    resolver.setLayoutUrl("layout/default.vm");
    return resolver;
}
```

72. Logging

Spring Boot has no mandatory logging dependency, 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.

72.1 Configure Logback for logging

If you put a `logback.xml` in the root of your classpath it will be picked up from there (or `logback-spring.xml` to take advantage of the templating features provided by Boot). 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 that `base.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.
- `${LOG_PATH}` if `logging.path` was set (representing a directory for log files to live in).
- `${LOG_EXCEPTION_CONVERSION_WORD}` if `logging.exception-conversion-word` was set in Boot's external configuration.

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).

Configure logback for file only output

If you want to disable console logging and write output only to a file you need a custom `logback-spring.xml` that imports `file-appender.xml` but not `console-appender.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml" />
    <property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-${java.io.tmpdir:-/
tmp}}/}spring.log}"/>
    <include resource="org/springframework/boot/logging/logback/file-appender.xml" />
    <root level="INFO">
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

You also need to add `logging.file` to your `application.properties`:

```
logging.file=myapplication.log
```

72.2 Configure Log4j for logging

Spring Boot supports Log4j 2 for logging configuration if it is on the classpath. If you are using the starter poms for assembling dependencies that means you have to exclude Logback and then include log4j 2 instead. If you aren't using the starter poms then you need to provide `commons-logging` (at least) in addition to Log4j 2.

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-log4j2</artifactId>
</dependency>
```

Note

The use 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 2). See the Actuator Log4j 2 samples for more detail and to see it in action.

Use YAML or JSON to configure Log4j 2

In addition to its default XML configuration format, Log4j 2 also supports YAML and JSON configuration files. To configure Log4j 2 to use an alternative configuration file format, add the appropriate dependencies to the classpath and name your configuration files to match your chosen file format:

Format	Dependencies	File names
YAML	`com.fasterxml.jackson.core:jackson-databind` `com.fasterxml.jackson.dataformat:jackson-dataformat-yaml`	`log4j2.yaml` `log4j2.yml`
JSON	`com.fasterxml.jackson.core:jackson-databind`	`log4j2.json` `log4j2.jsn`

73. Data Access

73.1 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 as explained in the section called “Third-party configuration”, 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 29.1, “Configure a DataSource”* in the ‘Spring Boot features’ section and the `DataSourceAutoConfiguration` class for more details.

Tip

You could also do that if you want to configure a JNDI data-source.

```
@Bean(destroyMethod="")
@ConfigurationProperties(prefix="datasource.mine")
public DataSource dataSource() throws Exception {
    JndiDataSourceLookup dataSourceLookup = new JndiDataSourceLookup();
    return dataSourceLookup.getDataSource("java:comp/env/jdbc/YourDS");
}
```

73.2 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();
}
```

73.3 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).

73.4 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 {

    //...

}
```

73.5 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.physical-strategy=com.example.MyPhysicalNamingStrategy
spring.jpa.database=H2
spring.jpa.show-sql=true
```

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.

73.6 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.

73.7 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.

If you are using Spring Data, you need to configure `@EnableJpaRepositories` accordingly:

```
@Configuration
@EnableJpaRepositories(basePackageClasses = Customer.class,
        entityManagerFactoryRef = "customerEntityManagerFactory")
public class CustomerConfiguration {
    ...
}

@Configuration
@EnableJpaRepositories(basePackageClasses = Order.class,
        entityManagerFactoryRef = "orderEntityManagerFactory")
public class OrderConfiguration {
    ...
}
```

73.8 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.

73.9 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.

73.10 Expose Spring Data repositories as REST endpoint

Spring Data REST can expose the `Repository` implementations as REST endpoints for you as long as Spring MVC has been enabled for the application.

Spring Boot exposes as set of useful properties from the `spring.data.rest` namespace that customize the `RepositoryRestConfiguration`. If you need to provide additional customization, you should use a `RepositoryRestConfigurer` bean.

74. Database initialization

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.

74.1 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.

74.2 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`, `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.

> **Note**
>
> You can output the schema creation by enabling the `org.hibernate.SQL` logger. This is done for you automatically if you enable the debug mode.

In addition, a file named `import.sql` in the root of the classpath 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).

74.3 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 fail-fast 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 fail-fast you can set `spring.datasource.continue-on-error=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.

74.4 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`.

74.5 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. Spring Boot will call `Flyway.migrate()` to perform the database migration. If you would like more control, provide a `@Bean` that implements `FlywayMigrationStrategy`.

> **Tip**
>
> If you want to make use of Flyway callbacks, those scripts should also live in the `classpath:db/migration` folder.

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.

75. Batch applications

75.1 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.

76. Actuator

76.1 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 46.3, “Customizing the management server port”* in the ‘Production-ready features’ section.

76.2 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 `server.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.

76.3 Actuator and Jersey

Actuator HTTP endpoints are only available for Spring MVC-based applications. If you want to use Jersey and still use the actuator you will need to enable Spring MVC (by depending on `spring-boot-starter-web`, for example). By default, both Jersey and the Spring MVC dispatcher servlet are mapped to the same path (`/`). You will need to change the path for one of them (by configuring `server.servlet-path` for Spring MVC or `spring.jersey.application-path` for Jersey). For example, if you add `server.servlet-path=/system` into `application.properties`, the actuator HTTP endpoints will be available under `/system`.

77. Security

77.1 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.

77.2 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.
    }

}
```

77.3 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.

78. Hot swapping

78.1 Reload static content

There are several options for hot reloading. The recommended approach is to use `spring-boot-devtools` as it provides additional development-time features such as support for fast application restarts and LiveReload as well as sensible development-time configuration (e.g. template caching).

Alternatively, running in an IDE (especially with debugging on) is a good way to do development (all modern IDEs allow reloading of static resources and usually also hot-swapping of Java class changes).

Finally, the Maven and Gradle plugins can be configured (see the `addResources` property) to support running from the command line with reloading of static files. You can use that with an external css/js compiler process if you are writing that code with higher level tools.

78.2 Reload templates without restarting the container

Most of the templating technologies supported by Spring Boot include a configuration option to disable caching (see below for details). If you're using the `spring-boot-devtools` module these properties will be automatically configured for you at development time.

Thymeleaf templates

If you are using Thymeleaf, then set `spring.thymeleaf.cache` to `false`. See `ThymeleafAutoConfiguration` for other Thymeleaf customization options.

FreeMarker templates

If you are using FreeMarker, then set `spring.freemarker.cache` to `false`. See `FreeMarkerAutoConfiguration` for other FreeMarker customization options.

Groovy templates

If you are using Groovy templates, then set `spring.groovy.template.cache` to `false`. See `GroovyTemplateAutoConfiguration` for other Groovy customization options.

Velocity templates

If you are using Velocity, then set `spring.velocity.cache` to `false`. See `VelocityAutoConfiguration` for other Velocity customization options.

78.3 Fast application restarts

The `spring-boot-devtools` module includes support for automatic application restarts. Whilst not as fast a technologies such as JRebel or Spring Loaded it's usually significantly faster than a "cold start". You should probably give it a try before investigating some of the more complex reload options discussed below.

For more details see the Chapter 20, *Developer tools* section.

78.4 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 IDEA 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 IDEA

You need to jump through a few hoops if you want to use Spring Loaded in combination with Gradle and IntelliJ IDEA. By default, IntelliJ IDEA will compile classes into a different location than Gradle, causing Spring Loaded monitoring to fail.

To configure IntelliJ IDEA correctly you can use the `idea` Gradle plugin:

```
buildscript {
    repositories { jcenter() }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-plugin:1.4.0.M2"
        classpath 'org.springframework:springloaded:1.2.0.RELEASE'
    }
}

apply plugin: 'idea'

idea {
    module {
        inheritOutputDirs = false
        outputDir = file("$buildDir/classes/main/")
    }
}

// ...
```

Note

IntelliJ IDEA 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 IDEA to automatically compile your code whenever a file is saved.

79. Build

79.1 Generate build information

Both the Maven and Gradle plugin allow to generate build information containing the coordinates, name and version of the project. The plugin can also be configured to add additional properties through configuration. When such file is present, Spring Boot auto-configures a `BuildProperties` bean.

To generate build information with Maven, add an execution for the `build-info` goal:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>1.4.0.M2</version>
            <executions>
                <execution>
                    <goals>
                        <goal>build-info</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Tip

Check the Spring Boot Maven Plugin documentation for more details.

And to do the same with Gradle:

```
springBoot  {
    buildInfo()
}
```

Additional properties can be added using the DSL:

```
springBoot  {
    buildInfo {
        additionalProperties = [
            'foo': 'bar'
        ]
    }
}
```

79.2 Generate git information

Both Maven and Gradle allow to generate a `git.properties` file containing information about the state of your `git` source code repository when the project was built.

For Maven users the `spring-boot-starter-parent` POM includes a pre-configured plugin to generate a `git.properties` file. 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>
```

Gradle users can achieve the same result using the `gradle-git-properties` plugin

```
plugins {
    id "com.gorylenko.gradle-git-properties" version "1.4.6"
}
```

79.3 Customize dependency versions

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.

To override dependency versions in Gradle, you can specify a version as shown below:

```
ext['slf4j.version'] = '1.7.5'
```

For additional information, please refer to the Gradle Dependency Management Plugin documentation.

79.4 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>1.4.0.M2</version>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

See the plugin documentation for full usage details.

79.5 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'
}
```

79.6 Extract specific libraries when an executable jar runs

Most nested libraries in an executable jar do not need to be unpacked in order to run, however, certain libraries can have problems. For example, JRuby includes its own nested jar support which assumes that the `jruby-complete.jar` is always directly available as a file in its own right.

To deal with any problematic libraries, you can flag that specific nested jars should be automatically unpacked 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']
}
```

79.7 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']
}
```

79.8 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.

79.9 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.

79.10 Build an executable archive from Ant without using spring-boot-antlib

To build with Ant you need to grab dependencies, compile and then create a jar or war archive. To make it executable you can either use the `spring-boot-antlib` module, or you can follow these instructions:

1. If you are building a jar, package the application's classes and resources in a nested `BOOT-INF/classes` directory. If you are building a war, package the application's classes in a nested `WEB-INF/classes` directory as usual.
2. Add the runtime dependencies in a nested `BOOT-INF/lib` directory for a jar or `WEB-INF/lib` for a war. Remember **not** to compress the entries in the archive.
3. Add the `provided` (embedded container) dependencies in a nested `BOOT-INF/lib` directory for jar or `WEB-INF/lib-provided` for a war. Remember **not** to compress the entries in the archive.
4. Add the `spring-boot-loader` classes at the root of the archive (so the `Main-Class` is available).
5. Use the appropriate launcher, e.g. `JarLauncher` for a jar file, as a `Main-Class` attribute in the manifest and specify the other properties it needs as manifest entries, principally a `Start-Class`.

Example:

```
<target name="build" depends="compile">
    <jar destfile="target/${ant.project.name}-${spring-boot.version}.jar" compress="false">
        <mappedresources>
            <fileset dir="target/classes" />
            <globmapper from="*" to="BOOT-INF/classes/*"/>
        </mappedresources>
        <mappedresources>
            <fileset dir="src/main/resources" erroronmissingdir="false"/>
            <globmapper from="*" to="BOOT-INF/classes/*"/>
        </mappedresources>
        <mappedresources>
            <fileset dir="${lib.dir}/runtime" />
            <globmapper from="*" to="BOOT-INF/lib/*"/>
        </mappedresources>
        <zipfileset src="${lib.dir}/loader/spring-boot-loader-jar-${spring-boot.version}.jar" />
        <manifest>
            <attribute name="Main-Class" value="org.springframework.boot.loader.JarLauncher" />
            <attribute name="Start-Class" value="${start-class}" />
        </manifest>
    </jar>
</target>
```

The Ant Sample has a `build.xml` with a `manual` task that should work if you run it with

```
$ ant -lib <folder containing ivy-2.2.jar> clean manual
```

after which you can run the application with

```
$ java -jar target/*.jar
```

79.11 How to use Java 6

If you want to use Spring Boot with Java 6 there are a small number of configuration changes that you will have to make. The exact changes depend on your application's functionality.

Embedded servlet container compatibility

If you are using one of Boot's embedded Servlet containers you will have to use a Java 6-compatible container. Both Tomcat 7 and Jetty 8 are Java 6 compatible. See Section 70.15, "Use Tomcat 7" and Section 70.16, "Use Jetty 8" for details.

JTA API compatibility

While the Java Transaction API itself doesn't require Java 7 the official API jar contains classes that have been built to require Java 7. If you are using JTA then you will need to replace the official JTA 1.2 API jar with one that has been built to work on Java 6. To do so, exclude any transitive dependencies on `javax.transaction:javax.transaction-api` and replace them with a dependency on `org.jboss.spec.javax.transaction:jboss-transaction-api_1.2_spec:1.0.0.Final`

80. Traditional deployment

80.1 Create a deployable war file

The first step in producing a deployable war file is to provide a `SpringBootServletInitializer` subclass and override its `configure` method. This makes use of Spring Framework's Servlet 3.0 support and allows you to configure your application when it's launched by the servlet container. Typically, you update your application's main class to extend `SpringBootServletInitializer`:

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }

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

}
```

The next step is to update your build configuration so that your project produces a war file rather than a jar file. If you're using Maven and using `spring-boot-starter-parent` (which configures Maven's war plugin for you) all you need to do is modify `pom.xml` to change the packaging to war:

```
<packaging>war</packaging>
```

If you're using Gradle, you need to modify `build.gradle` to apply the war plugin to the project:

```
apply plugin: 'war'
```

The final step in the process is to ensure that the embedded servlet container doesn't interfere with the servlet container to which the war file will be deployed. To do so, you need to mark the embedded servlet container dependency as provided.

If you're using Maven:

```
<dependencies>
    <!-- … -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <!-- … -->
</dependencies>
```

And if you're using Gradle:

```
dependencies {
    // …
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    // …
}
```

If you're using the Spring Boot build tools, marking the embedded servlet container dependency as provided will produce an executable war file with the provided dependencies packaged in a `lib-`

`provided` directory. This means that, in addition to being deployable to a servlet container, you can also run your application using `java -jar` on the command line.

> **Tip**
>
> Take a look at Spring Boot's sample applications for a Maven-based example of the above-described configuration.

80.2 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`.

80.3 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.

80.4 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>
```

80.5 Deploying a WAR in an Old (Servlet 2.5) Container

Spring Boot uses Servlet 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>
    </filter>

    <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
# ----------------------------------------

# BANNER
banner.charset=UTF-8 # Banner file encoding.
banner.location=classpath:banner.txt # Banner file location.
banner.image.location=classpath:banner.gif # (Banner image file location, jpg/png can also be used).
banner.image.width= # Width of the banner image in chars (default 76)
banner.image.height= # Height of the banner image in chars (default based on image height)
banner.image.margin= # Left hand image margin in chars (default 2)
banner.image.invert= # If images should be inverted for dark terminal themes (default false)

# LOGGING
logging.config= # Location of the logging configuration file. For instance `classpath:logback.xml` for
 Logback
logging.exception-conversion-word=%wEx # Conversion word used when logging exceptions.
logging.file= # Log file name. For instance `myapp.log`
logging.level.*= # Log levels severity mapping. For instance `logging.level.org.springframework=DEBUG`
logging.path= # Location of the log file. For instance `/var/log`
logging.pattern.console= # Appender pattern for output to the console. Only supported with the default
 logback setup.
logging.pattern.file= # Appender pattern for output to the file. Only supported with the default logback
 setup.
logging.pattern.level= # Appender pattern for log level (default %5p). Only supported with the default
 logback setup.
logging.register-shutdown-hook=false # Register a shutdown hook for the logging system when it is
 initialized.

# AOP
spring.aop.auto=true # Add @EnableAspectJAutoProxy.
spring.aop.proxy-target-class=false # Whether subclass-based (CGLIB) proxies are to be created (true) as
 opposed to standard Java interface-based proxies (false).

# IDENTITY (ContextIdApplicationContextInitializer)
spring.application.index= # Application index.
spring.application.name= # Application name.

# ADMIN (SpringApplicationAdminJmxAutoConfiguration)
spring.application.admin.enabled=false # Enable admin features for the application.
spring.application.admin.jmx-name=org.springframework.boot:type=Admin,name=SpringApplication # JMX name
 of the application admin MBean.

# AUTO-CONFIGURATION
spring.autoconfigure.exclude= # Auto-configuration classes to exclude.

# SPRING CORE
spring.beaninfo.ignore=true # Skip search of BeanInfo classes.

# SPRING CACHE (CacheProperties)
spring.cache.cache-names= # Comma-separated list of cache names to create if supported by the underlying
 cache manager.
spring.cache.caffeine.spec= # The spec to use to create caches. Check CaffeineSpec for more details on
 the spec format.
spring.cache.couchbase.expiration=0 # Entry expiration in milliseconds. By default the entries never
 expire.
spring.cache.ehcache.config= # The location of the configuration file to use to initialize EhCache.
spring.cache.guava.spec= # The spec to use to create caches. Check CacheBuilderSpec for more details on
 the spec format.
spring.cache.hazelcast.config= # The location of the configuration file to use to initialize Hazelcast.
spring.cache.infinispan.config= # The location of the configuration file to use to initialize
 Infinispan.
spring.cache.jcache.config= # The location of the configuration file to use to initialize the cache
 manager.
spring.cache.jcache.provider= # Fully qualified name of the CachingProvider implementation to use to
 retrieve the JSR-107 compliant cache manager. Only needed if more than one JSR-107 implementation is
 available on the classpath.
spring.cache.type= # Cache type, auto-detected according to the environment by default.

# SPRING CONFIG - using environment property only (ConfigFileApplicationListener)
spring.config.location= # Config file locations.
spring.config.name=application # Config file name.

# HAZELCAST (HazelcastProperties)
spring.hazelcast.config= # The location of the configuration file to use to initialize Hazelcast.

# PROJECT INFORMATION (ProjectInfoProperties)
spring.info.build.location=classpath:META-INF/boot/build.properties # Location of the generated
 build.properties file.
spring.info.git.location=classpath:git.properties # Location of the generated git.properties file.

# JMX
spring.jmx.default-domain= # JMX domain name.
spring.jmx.enabled=true # Expose management beans to the JMX domain.
spring.jmx.server=mbeanServer # MBeanServer bean name.

# Email (MailProperties)
spring.mail.default-encoding=UTF-8 # Default MimeMessage encoding.
spring.mail.host= # SMTP server host. For instance `smtp.example.com`
spring.mail.jndi-name= # Session JNDI name. When set, takes precedence to others mail settings.
spring.mail.password= # Login password of the SMTP server.
spring.mail.port= # SMTP server port.
spring.mail.properties.*= # Additional JavaMail session properties.
spring.mail.protocol=smtp # Protocol used by the SMTP server.
spring.mail.test-connection=false # Test that the mail server is available on startup.
spring.mail.username= # Login user of the SMTP server.

# APPLICATION SETTINGS (SpringApplication)
spring.main.banner-mode=console # Mode used to display the banner when the application runs.
spring.main.sources= # Sources (class name, package name or XML resource location) to include in the
 ApplicationContext.
spring.main.web-environment= # Run the application in a web environment (auto-detected by default).

# FILE ENCODING (FileEncodingApplicationListener)
spring.mandatory-file-encoding= # Expected character encoding the application must use.

# INTERNATIONALIZATION (MessageSourceAutoConfiguration)
```

```
spring.messages.always-use-message-format=false # Set whether to always apply the MessageFormat rules,
 parsing even messages without arguments.
spring.messages.basename=messages # Comma-separated list of basenames, each following the ResourceBundle
 convention.
spring.messages.cache-seconds=-1 # Loaded resource bundle files cache expiration, in seconds. When set
 to -1, bundles are cached forever.
spring.messages.encoding=UTF-8 # Message bundles encoding.
spring.messages.fallback-to-system-locale=true # Set whether to fall back to the system Locale if no
 files for a specific Locale have been found.

# OUTPUT
spring.output.ansi.enabled=detect # Configure the ANSI output (can be "detect", "always", "never").

# PID FILE (ApplicationPidFileWriter)
spring.pid.fail-on-write-error= # Fail if ApplicationPidFileWriter is used but it cannot write the PID
 file.
spring.pid.file= # Location of the PID file to write (if ApplicationPidFileWriter is used).

# PROFILES
spring.profiles.active= # Comma-separated list of active profiles.
spring.profiles.include= # Unconditionally activate the specified comma separated profiles.

# SENDGRID (SendGridAutoConfiguration)
spring.sendgrid.api-key= # SendGrid api key (alternative to username/password)
spring.sendgrid.username= # SendGrid account username
spring.sendgrid.password= # SendGrid account password
spring.sendgrid.proxy.host= # SendGrid proxy host
spring.sendgrid.proxy.port= # SendGrid proxy port


# ----------------------------------------
# WEB PROPERTIES
# ----------------------------------------

# MULTIPART (MultipartProperties)
multipart.enabled=true # Enable support of multi-part uploads.
multipart.file-size-threshold=0 # Threshold after which files will be written to disk. Values can use
 the suffixed "MB" or "KB" to indicate a Megabyte or Kilobyte size.
multipart.location= # Intermediate location of uploaded files.
multipart.max-file-size=1Mb # Max file size. Values can use the suffixed "MB" or "KB" to indicate a
 Megabyte or Kilobyte size.
multipart.max-request-size=10Mb # Max request size. Values can use the suffixed "MB" or "KB" to indicate
 a Megabyte or Kilobyte size.

# EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.address= # Network address to which the server should bind to.
server.compression.enabled=false # If response compression is enabled.
server.compression.excluded-user-agents= # List of user-agents to exclude from compression.
server.compression.mime-types= # Comma-separated list of MIME types that should be compressed. For
 instance `text/html,text/css,application/json`
server.compression.min-response-size= # Minimum response size that is required for compression to be
 performed. For instance 2048
server.context-parameters.*= # Servlet context init parameters. For instance `server.context-
parameters.a=alpha`
server.context-path= # Context path of the application.
server.display-name=application # Display name of the application.
server.max-http-header-size=0 # Maximum size in bytes of the HTTP message header.
server.max-http-post-size=0 # Maximum size in bytes of the HTTP post content.
server.error.include-stacktrace=never # When to include a "stacktrace" attribute.
server.error.path=/error # Path of the error controller.
server.error.whitelabel.enabled=true # Enable the default error page displayed in browsers in case of a
 server error.
server.jetty.acceptors= # Number of acceptor threads to use.
server.jetty.selectors= # Number of selector threads to use.
server.jsp-servlet.class-name=org.apache.jasper.servlet.JspServlet # The class name of the JSP servlet.
server.jsp-servlet.init-parameters.*= # Init parameters used to configure the JSP servlet
server.jsp-servlet.registered=true # Whether or not the JSP servlet is registered
server.port=8080 # Server HTTP port.
server.server-header= # The value sent in the server response header (uses servlet container default if
 empty)
server.servlet-path=/ # Path of the main dispatcher servlet.
server.use-forward-headers= # If X-Forwarded-* headers should be applied to the HttpRequest.
server.session.cookie.comment= # Comment for the session cookie.
server.session.cookie.domain= # Domain for the session cookie.
server.session.cookie.http-only= # "HttpOnly" flag for the session cookie.
server.session.cookie.max-age= # Maximum age of the session cookie in seconds.
server.session.cookie.name= # Session cookie name.
server.session.cookie.path= # Path of the session cookie.
server.session.cookie.secure= # "Secure" flag for the session cookie.
server.session.persistent=false # Persist session data between restarts.
server.session.store-dir= # Directory used to store session data.
server.session.timeout= # Session timeout in seconds.
server.session.tracking-modes= # Session tracking modes (one or more of the following: "cookie", "url",
 "ssl").
server.ssl.ciphers= # Supported SSL ciphers.
server.ssl.client-auth= # Whether client authentication is wanted ("want") or needed ("need"). Requires
 a trust store.
server.ssl.enabled= # Enable SSL support.
server.ssl.enabled-protocols= # Enabled SSL protocols.
server.ssl.key-alias= # Alias that identifies the key in the key store.
server.ssl.key-password= # Password used to access the key in the key store.
server.ssl.key-store= # Path to the key store that holds the SSL certificate (typically a jks file).
server.ssl.key-store-password= # Password used to access the key store.
server.ssl.key-store-provider= # Provider for the key store.
server.ssl.key-store-type= # Type of the key store.
server.ssl.protocol=TLS # SSL protocol to use.
server.ssl.trust-store= # Trust store that holds SSL certificates.
server.ssl.trust-store-password= # Password used to access the trust store.
server.ssl.trust-store-provider= # Provider for the trust store.
server.ssl.trust-store-type= # Type of the trust store.
server.tomcat.accesslog.directory=logs # Directory in which log files are created. Can be relative to
 the tomcat base dir or absolute.
server.tomcat.accesslog.enabled=false # Enable access log.
server.tomcat.accesslog.pattern=common # Format pattern for access logs.
server.tomcat.accesslog.prefix=access_log # Log file name prefix.
server.tomcat.accesslog.suffix=.log # Log file name suffix.
server.tomcat.background-processor-delay=30 # Delay in seconds between the invocation of
 backgroundProcess methods.
server.tomcat.basedir= # Tomcat base directory. If not specified a temporary directory will be used.
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}|\\
        172\\.1[6-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\
        172\\.2[0-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\
        172\\.3[0-1]{1}\\.\\d{1,3}\\.\\d{1,3} # regular expression matching trusted IP addresses.
server.tomcat.max-threads=0 # Maximum amount of worker threads.
server.tomcat.min-spare-threads=0 # Minimum amount of worker threads.
server.tomcat.port-header=X-Forwarded-Port # Name of the HTTP header used to override the original port
 value.
server.tomcat.protocol-header= # Header that holds the incoming protocol, usually named "X-Forwarded-
Proto".
server.tomcat.protocol-header-https-value=https # Value of the protocol header that indicates that the
 incoming request uses SSL.
server.tomcat.remote-ip-header= # Name of the http header from which the remote ip is extracted. For
 instance `X-FORWARDED-FOR`
server.tomcat.uri-encoding=UTF-8 # Character encoding to use to decode the URI.
server.undertow.accesslog.dir= # Undertow access log directory.
server.undertow.accesslog.enabled=false # Enable access log.
server.undertow.accesslog.pattern=common # Format pattern for access logs.
server.undertow.buffer-size= # Size of each buffer in bytes.
server.undertow.buffers-per-region= # Number of buffer per region.
server.undertow.direct-buffers= # Allocate buffers outside the Java heap.
server.undertow.io-threads= # Number of I/O threads to create for the worker.
server.undertow.worker-threads= # Number of worker threads.

# FREEMARKER (FreeMarkerAutoConfiguration)
spring.freemarker.allow-request-override=false # Set whether HttpServletRequest attributes are allowed
 to override (hide) controller generated model attributes of the same name.
spring.freemarker.allow-session-override=false # Set whether HttpSession attributes are allowed to
 override (hide) controller generated model attributes of the same name.
spring.freemarker.cache=false # Enable template caching.
spring.freemarker.charset=UTF-8 # Template encoding.
spring.freemarker.check-template-location=true # Check that the templates location exists.
spring.freemarker.content-type=text/html # Content-Type value.
spring.freemarker.enabled=true # Enable MVC view resolution for this technology.
spring.freemarker.expose-request-attributes=false # Set whether all request attributes should be added
 to the model prior to merging with the template.
spring.freemarker.expose-session-attributes=false # Set whether all HttpSession attributes should be
 added to the model prior to merging with the template.
spring.freemarker.expose-spring-macro-helpers=true # Set whether to expose a RequestContext for use by
 Spring's macro library, under the name "springMacroRequestContext".
spring.freemarker.prefer-file-system-access=true # Prefer file system access for template loading. File
 system access enables hot detection of template changes.
spring.freemarker.prefix= # Prefix that gets prepended to view names when building a URL.
spring.freemarker.request-context-attribute= # Name of the RequestContext attribute for all views.
spring.freemarker.settings.*= # Well-known FreeMarker keys which will be passed to FreeMarker's
 Configuration.
spring.freemarker.suffix= # Suffix that gets appended to view names when building a URL.
spring.freemarker.template-loader-path=classpath:/templates/ # Comma-separated list of template paths.
spring.freemarker.view-names= # White list of view names that can be resolved.

# GROOVY TEMPLATES (GroovyTemplateAutoConfiguration)
spring.groovy.template.allow-request-override=false # Set whether HttpServletRequest attributes are
 allowed to override (hide) controller generated model attributes of the same name.
spring.groovy.template.allow-session-override=false # Set whether HttpSession attributes are allowed to
 override (hide) controller generated model attributes of the same name.
spring.groovy.template.cache= # Enable template caching.
spring.groovy.template.charset=UTF-8 # Template encoding.
spring.groovy.template.check-template-location=true # Check that the templates location exists.
spring.groovy.template.configuration.*= # See GroovyMarkupConfigurer
spring.groovy.template.content-type=test/html # Content-Type value.
spring.groovy.template.enabled=true # Enable MVC view resolution for this technology.
spring.groovy.template.expose-request-attributes=false # Set whether all request attributes should be
 added to the model prior to merging with the template.
spring.groovy.template.expose-session-attributes=false # Set whether all HttpSession attributes should
 be added to the model prior to merging with the template.
spring.groovy.template.expose-spring-macro-helpers=true # Set whether to expose a RequestContext for use
 by Spring's macro library, under the name "springMacroRequestContext".
spring.groovy.template.prefix= # Prefix that gets prepended to view names when building a URL.
spring.groovy.template.request-context-attribute= # Name of the RequestContext attribute for all views.
spring.groovy.template.resource-loader-path=classpath:/templates/ # Template path.
spring.groovy.template.suffix=.tpl # Suffix that gets appended to view names when building a URL.
spring.groovy.template.view-names= # White list of view names that can be resolved.

# SPRING HATEOAS (HateoasProperties)
spring.hateoas.use-hal-as-default-json-media-type=true # Specify if application/hal+json responses
 should be sent to requests that accept application/json.

# HTTP message conversion
spring.http.converters.preferred-json-mapper=jackson # Preferred JSON mapper to use for HTTP message
 conversion. Set to "gson" to force the use of Gson when both it and Jackson are on the classpath.

# HTTP encoding (HttpEncodingProperties)
spring.http.encoding.charset=UTF-8 # Charset of HTTP requests and responses. Added to the "Content-Type"
 header if not set explicitly.
spring.http.encoding.enabled=true # Enable http encoding support.
spring.http.encoding.force=true # Force the encoding to the configured charset on HTTP requests and
 responses.

# JACKSON (JacksonProperties)
spring.jackson.date-format= # Date format string or a fully-qualified date format class name. For
 instance `yyyy-MM-dd HH:mm:ss`.
spring.jackson.default-property-inclusion= # Controls the inclusion of properties during serialization.
spring.jackson.deserialization.*= # Jackson on/off features that affect the way Java objects are
 deserialized.
spring.jackson.generator.*= # Jackson on/off features for generators.
spring.jackson.joda-date-time-format= # Joda date time format string. If not configured, "date-format"
 will be used as a fallback if it is configured with a format string.
spring.jackson.locale= # Locale used for formatting.
spring.jackson.mapper.*= # Jackson general purpose on/off features.
spring.jackson.parser.*= # Jackson on/off features for parsers.
spring.jackson.property-naming-strategy= # One of the constants on Jackson's PropertyNamingStrategy. Can
 also be a fully-qualified class name of a PropertyNamingStrategy subclass.
spring.jackson.serialization.*= # Jackson on/off features that affect the way Java objects are
 serialized.
spring.jackson.serialization-inclusion= # Controls the inclusion of properties during serialization.
 Configured with one of the values in Jackson's JsonInclude.Include enumeration.
spring.jackson.time-zone= # Time zone used when formatting dates. For instance `America/Los_Angeles`

# JERSEY (JerseyProperties)
spring.jersey.application-path= # Path that serves as the base URI for the application. Overrides the
 value of "@ApplicationPath" if specified.
spring.jersey.filter.order=0 # Jersey filter chain order.
spring.jersey.init.*= # Init parameters to pass to Jersey via the servlet or filter.
spring.jersey.servlet.load-on-startup=-1 # Load on startup priority of the Jersey servlet.
spring.jersey.type=servlet # Jersey integration type. Can be either "servlet" or "filter".

# SPRING MOBILE DEVICE VIEWS (DeviceDelegatingViewResolverAutoConfiguration)
spring.mobile.devicedelegatingviewresolver.enable-fallback=false # Enable support for fallback
 resolution.
spring.mobile.devicedelegatingviewresolver.enabled=false # Enable device view resolver.
spring.mobile.devicedelegatingviewresolver.mobile-prefix=mobile/ # Prefix that gets prepended to view
 names for mobile devices.
spring.mobile.devicedelegatingviewresolver.mobile-suffix= # Suffix that gets appended to view names for
 mobile devices.
spring.mobile.devicedelegatingviewresolver.normal-prefix= # Prefix that gets prepended to view names for
 normal devices.
spring.mobile.devicedelegatingviewresolver.normal-suffix= # Suffix that gets appended to view names for
 normal devices.
spring.mobile.devicedelegatingviewresolver.tablet-prefix=tablet/ # Prefix that gets prepended to view
 names for tablet devices.
spring.mobile.devicedelegatingviewresolver.tablet-suffix= # Suffix that gets appended to view names for
 tablet devices.

# SPRING MOBILE SITE PREFERENCE (SitePreferenceAutoConfiguration)
spring.mobile.sitepreference.enabled=true # Enable SitePreferenceHandler.

# MUSTACHE TEMPLATES (MustacheAutoConfiguration)
spring.mustache.allow-request-override= # Set whether HttpServletRequest attributes are allowed to
 override (hide) controller generated model attributes of the same name.
spring.mustache.allow-session-override= # Set whether HttpSession attributes are allowed to override
 (hide) controller generated model attributes of the same name.
spring.mustache.cache= # Enable template caching.
spring.mustache.charset= # Template encoding.
spring.mustache.check-template-location= # Check that the templates location exists.
spring.mustache.content-type= # Content-Type value.
spring.mustache.enabled= # Enable MVC view resolution for this technology.
spring.mustache.expose-request-attributes= # Set whether all request attributes should be added to the
 model prior to merging with the template.
spring.mustache.expose-session-attributes= # Set whether all HttpSession attributes should be added to
 the model prior to merging with the template.
spring.mustache.expose-spring-macro-helpers= # Set whether to expose a RequestContext for use by
 Spring's macro library, under the name "springMacroRequestContext".
spring.mustache.prefix=classpath:/templates/ # Prefix to apply to template names.
spring.mustache.request-context-attribute= # Name of the RequestContext attribute for all views.
spring.mustache.suffix=.html # Suffix to apply to template names.
spring.mustache.view-names= # White list of view names that can be resolved.

# SPRING MVC (WebMvcProperties)
spring.mvc.async.request-timeout= # Amount of time (in milliseconds) before asynchronous request
 handling times out.
spring.mvc.date-format= # Date format to use. For instance `dd/MM/yyyy`.
spring.mvc.dispatch-trace-request=false # Dispatch TRACE requests to the FrameworkServlet doService
 method.
spring.mvc.dispatch-options-request=false # Dispatch OPTIONS requests to the FrameworkServlet doService
 method.
spring.mvc.favicon.enabled=true # Enable resolution of favicon.ico.
spring.mvc.ignore-default-model-on-redirect=true # If the content of the "default" model should be
 ignored during redirect scenarios.
spring.mvc.locale= # Locale to use.
spring.mvc.media-types.*= # Maps file extensions to media types for content negotiation.
spring.mvc.message-codes-resolver-format= # Formatting strategy for message codes. For instance
 `PREFIX_ERROR_CODE`.
spring.mvc.static-path-pattern=/** # Path pattern used for static resources.
spring.mvc.throw-exception-if-no-handler-found=false # If a "NoHandlerFoundException" should be thrown
 if no Handler was found to process a request.
spring.mvc.view.prefix= # Spring MVC view prefix.
spring.mvc.view.suffix= # Spring MVC view suffix.

# SPRING RESOURCES HANDLING (ResourceProperties)
spring.resources.add-mappings=true # Enable default resource handling.
spring.resources.cache-period= # Cache period for the resources served by the resource handler, in
 seconds.
spring.resources.chain.cache=true # Enable caching in the Resource chain.
spring.resources.chain.enabled= # Enable the Spring Resource Handling chain. Disabled by default unless
 at least one strategy has been enabled.
spring.resources.chain.gzipped=false # Enable resolution of already gzipped resources.
spring.resources.chain.html-application-cache=false # Enable HTML5 application cache manifest rewriting.
spring.resources.chain.strategy.content.enabled=false # Enable the content Version Strategy.
spring.resources.chain.strategy.content.paths=/** # Comma-separated list of patterns to apply to the
 Version Strategy.
spring.resources.chain.strategy.fixed.enabled=false # Enable the fixed Version Strategy.
spring.resources.chain.strategy.fixed.paths= # Comma-separated list of patterns to apply to the Version
 Strategy.
spring.resources.chain.strategy.fixed.version= # Version string to use for the Version Strategy.
spring.resources.static-locations=classpath:/META-INF/resources/,classpath:/resources/,classpath:/
static/,classpath:/public/ # Locations of static resources.

# SPRING SOCIAL (SocialWebAutoConfiguration)
spring.social.auto-connection-views=false # Enable the connection status view for supported providers.

# 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

# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.cache=true # Enable template caching.
spring.thymeleaf.check-template-location=true # Check that the templates location exists.
spring.thymeleaf.content-type=text/html # Content-Type value.
spring.thymeleaf.enabled=true # Enable MVC Thymeleaf view resolution.
spring.thymeleaf.encoding=UTF-8 # Template encoding.
spring.thymeleaf.excluded-view-names= # Comma-separated list of view names that should be excluded from
 resolution.
spring.thymeleaf.mode=HTML5 # Template mode to be applied to templates. See also
 StandardTemplateModeHandlers.
spring.thymeleaf.prefix=classpath:/templates/ # Prefix that gets prepended to view names when building a
 URL.
spring.thymeleaf.suffix=.html # Suffix that gets appended to view names when building a URL.
spring.thymeleaf.template-resolver-order= # Order of the template resolver in the chain.
spring.thymeleaf.view-names= # Comma-separated list of view names that can be resolved.

# VELOCITY TEMPLATES (VelocityAutoConfiguration)
spring.velocity.allow-request-override=false # Set whether HttpServletRequest attributes are allowed to
 override (hide) controller generated model attributes of the same name.
spring.velocity.allow-session-override=false # Set whether HttpSession attributes are allowed to
 override (hide) controller generated model attributes of the same name.
spring.velocity.cache= # Enable template caching.
spring.velocity.charset=UTF-8 # Template encoding.
spring.velocity.check-template-location=true # Check that the templates location exists.
spring.velocity.content-type=text/html # Content-Type value.
spring.velocity.date-tool-attribute= # Name of the DateTool helper object to expose in the Velocity
 context of the view.
spring.velocity.enabled=true # Enable MVC view resolution for this technology.
spring.velocity.expose-request-attributes=false # Set whether all request attributes should be added to
 the model prior to merging with the template.
spring.velocity.expose-session-attributes=false # Set whether all HttpSession attributes should be added
 to the model prior to merging with the template.
spring.velocity.expose-spring-macro-helpers=true # Set whether to expose a RequestContext for use by
 Spring's macro library, under the name "springMacroRequestContext".
spring.velocity.number-tool-attribute= # Name of the NumberTool helper object to expose in the Velocity
 context of the view.
spring.velocity.prefer-file-system-access=true # Prefer file system access for template loading. File
 system access enables hot detection of template changes.
spring.velocity.prefix= # Prefix that gets prepended to view names when building a URL.
spring.velocity.properties.*= # Additional velocity properties.
spring.velocity.request-context-attribute= # Name of the RequestContext attribute for all views.
spring.velocity.resource-loader-path=classpath:/templates/ # Template path.
spring.velocity.suffix=.vm # Suffix that gets appended to view names when building a URL.
spring.velocity.toolbox-config-location= # Velocity Toolbox config location. For instance `/WEB-INF/
toolbox.xml`
spring.velocity.view-names= # White list of view names that can be resolved.



# ----------------------------------------
# SECURITY PROPERTIES
# ----------------------------------------
# SECURITY (SecurityProperties)
security.basic.authorize-mode=role # Security authorize mode to apply.
security.basic.enabled=true # Enable basic authentication.
security.basic.path=/** # Comma-separated list of paths to secure.
security.basic.realm=Spring # HTTP basic realm name.
security.enable-csrf=false # Enable Cross Site Request Forgery support.
security.filter-order=0 # Security filter chain order.
security.filter-dispatcher-types=ASYNC, FORWARD, INCLUDE, REQUEST # Security filter chain dispatcher
 types.
security.headers.cache=true # Enable cache control HTTP headers.
security.headers.content-type=true # Enable "X-Content-Type-Options" header.
security.headers.frame=true # Enable "X-Frame-Options" header.
security.headers.hsts= # HTTP Strict Transport Security (HSTS) mode (none, domain, all).
security.headers.xss=true # Enable cross site scripting (XSS) protection.
security.ignored= # Comma-separated list of paths to exclude from the default secured paths.
security.require-ssl=false # Enable secure channel for all requests.
security.sessions=stateless # Session creation policy (always, never, if_required, stateless).
security.user.name=user # Default user name.
security.user.password= # Password for the default user name. A random password is logged on startup by
 default.
security.user.role=USER # Granted roles for the default user name.

# SECURITY OAUTH2 CLIENT (OAuth2ClientProperties
security.oauth2.client.client-id= # OAuth2 client id.
security.oauth2.client.client-secret= # OAuth2 client secret. A random secret is generated by default

# SECURITY OAUTH2 RESOURCES (ResourceServerProperties
security.oauth2.resource.id= # Identifier of the resource.
security.oauth2.resource.jwt.key-uri= # The URI of the JWT token. Can be set if the value is not
 available and the key is public.
security.oauth2.resource.jwt.key-value= # The verification key of the JWT token. Can either be a
 symmetric secret or PEM-encoded RSA public key.
security.oauth2.resource.prefer-token-info=true # Use the token info, can be set to false to use the
 user info.
security.oauth2.resource.service-id=resource #
security.oauth2.resource.token-info-uri= # URI of the token decoding endpoint.
security.oauth2.resource.token-type= # The token type to send when using the userInfoUri.
security.oauth2.resource.user-info-uri= # URI of the user endpoint.

# SECURITY OAUTH2 SSO (OAuth2SsoProperties
security.oauth2.sso.filter-order= # Filter order to apply if not providing an explicit
 WebSecurityConfigurerAdapter
security.oauth2.sso.login-path=/login # Path to the login page, i.e. the one that triggers the redirect
 to the OAuth2 Authorization Server


# ----------------------------------------
# DATA PROPERTIES
# ----------------------------------------

# FLYWAY (FlywayProperties)
flyway.baseline-description= #
flyway.baseline-version=1 # version to start migration
flyway.baseline-on-migrate= #
flyway.check-location=false # Check that migration scripts location exists.
flyway.clean-on-validation-error= #
flyway.enabled=true # Enable flyway.
flyway.encoding= #
flyway.ignore-failed-future-migration= #
flyway.init-sqls= # SQL statements to execute to initialize a connection immediately after obtaining it.
flyway.locations=classpath:db/migration # locations of migrations scripts
flyway.out-of-order= #
flyway.password= # JDBC password if you want Flyway to create its own DataSource
flyway.placeholder-prefix= #
flyway.placeholder-replacement= #
flyway.placeholder-suffix= #
flyway.placeholders.*= #
flyway.schemas= # schemas to update
flyway.sql-migration-prefix=V #
flyway.sql-migration-separator= #
flyway.sql-migration-suffix=.sql #
flyway.table= #
flyway.url= # JDBC url of the database to migrate. If not set, the primary configured data source is
 used.
flyway.user= # Login user of the database to migrate.
flyway.validate-on-migrate= #

# LIQUIBASE (LiquibaseProperties)
liquibase.change-log=classpath:/db/changelog/db.changelog-master.yaml # Change log configuration path.
liquibase.check-change-log-location=true # Check the change log location exists.
liquibase.contexts= # Comma-separated list of runtime contexts to use.
liquibase.default-schema= # Default database schema.
liquibase.drop-first=false # Drop the database schema first.
liquibase.enabled=true # Enable liquibase support.
liquibase.labels= # Comma-separated list of runtime labels to use.
liquibase.parameters.*= # Change log parameters.
liquibase.password= # Login password of the database to migrate.
liquibase.rollback-file= # File to which rollback SQL will be written when an update is performed.
liquibase.url= # JDBC url of the database to migrate. If not set, the primary configured data source is
 used.
liquibase.user= # Login user of the database to migrate.

# COUCHBASE (CouchbaseProperties)
spring.couchbase.bootstrap-hosts= # Couchbase nodes (host or IP address) to bootstrap from.
spring.couchbase.bucket.name=default # Name of the bucket to connect to.
spring.couchbase.bucket.password=  # Password of the bucket.
spring.couchbase.env.endpoints.key-value=1 # Number of sockets per node against the Key/value service.
spring.couchbase.env.endpoints.query=1 # Number of sockets per node against the Query (N1QL) service.
spring.couchbase.env.endpoints.view=1 # Number of sockets per node against the view service.
spring.couchbase.env.ssl.enabled= # Enable SSL support. Enabled automatically if a "keyStore" is
 provided unless specified otherwise.
spring.couchbase.env.ssl.key-store= # Path to the JVM key store that holds the certificates.
spring.couchbase.env.ssl.key-store-password= # Password used to access the key store.
spring.couchbase.env.timeouts.connect=5000 # Bucket connections timeout in milliseconds.
spring.couchbase.env.timeouts.key-value=2500 # Blocking operations performed on a specific key timeout
 in milliseconds.
spring.couchbase.env.timeouts.query=7500 # N1QL query operations timeout in milliseconds.
spring.couchbase.env.timeouts.socket-connect=1000 # Socket connect connections timeout in milliseconds.
spring.couchbase.env.timeouts.view=7500 # Regular and geospatial view operations timeout in
 milliseconds.

# DAO (PersistenceExceptionTranslationAutoConfiguration)
spring.dao.exceptiontranslation.enabled=true # Enable the PersistenceExceptionTranslationPostProcessor.

# CASSANDRA (CassandraProperties)
spring.data.cassandra.cluster-name= # Name of the Cassandra cluster.
spring.data.cassandra.compression= # Compression supported by the Cassandra binary protocol.
spring.data.cassandra.connect-timeout-millis= # Socket option: connection time out.
spring.data.cassandra.consistency-level= # Queries consistency level.
spring.data.cassandra.contact-points=localhost # Comma-separated list of cluster node addresses.
spring.data.cassandra.fetch-size= # Queries default fetch size.
spring.data.cassandra.keyspace-name= # Keyspace name to use.
spring.data.cassandra.load-balancing-policy= # Class name of the load balancing policy.
spring.data.cassandra.port= # Port of the Cassandra server.
spring.data.cassandra.password= # Login password of the server.
spring.data.cassandra.read-timeout-millis= # Socket option: read time out.
spring.data.cassandra.reconnection-policy= # Reconnection policy class.
spring.data.cassandra.retry-policy= # Class name of the retry policy.
spring.data.cassandra.serial-consistency-level= # Queries serial consistency level.
spring.data.cassandra.ssl=false # Enable SSL support.
spring.data.cassandra.username= # Login user of the server.

# DATA COUCHBASE (CouchbaseDataProperties)
spring.data.couchbase.auto-index=false # Automatically create views and indexes.
spring.data.couchbase.consistency=read-your-own-writes # Consistency to apply by default on generated
 queries.
spring.data.couchbase.repositories.enabled=true # Enable Couchbase repositories.

# ELASTICSEARCH (ElasticsearchProperties)
spring.data.elasticsearch.cluster-name=elasticsearch # Elasticsearch cluster name.
spring.data.elasticsearch.cluster-nodes= # Comma-separated list of cluster node addresses. If not
 specified, starts a client node.
spring.data.elasticsearch.properties.*= # Additional properties used to configure the client.
spring.data.elasticsearch.repositories.enabled=true # Enable Elasticsearch repositories.

# MONGODB (MongoProperties)
spring.data.mongodb.authentication-database= # Authentication database name.
spring.data.mongodb.database=test # Database name.
spring.data.mongodb.field-naming-strategy= # Fully qualified name of the FieldNamingStrategy to use.
spring.data.mongodb.grid-fs-database= # GridFS database name.
spring.data.mongodb.host=localhost # Mongo server host.
spring.data.mongodb.password= # Login password of the mongo server.
spring.data.mongodb.port=27017 # Mongo server port.
spring.data.mongodb.repositories.enabled=true # Enable Mongo repositories.
spring.data.mongodb.uri=mongodb://localhost/test # Mongo database URI. When set, host and port are
 ignored.
spring.data.mongodb.username= # Login user of the mongo server.

# DATA REDIS
spring.data.redis.repositories.enabled=true # Enable Redis repositories.

# NEO4J (Neo4jProperties)
spring.data.neo4j.compiler= # Compiler to use.
spring.data.neo4j.embedded.enabled=true # Enable embedded mode if the embedded driver is available.
spring.data.neo4j.password= # Login password of the server.
spring.data.neo4j.repositories.enabled=true # Enable Neo4j repositories.
spring.data.neo4j.session.scope=singleton # Scope (lifetime) of the session.
spring.data.neo4j.uri= # URI used by the driver. Auto-detected by default.
spring.data.neo4j.username= # Login user of the server.

# DATA REST (RepositoryRestProperties)
spring.data.rest.base-path= # Base path to be used by Spring Data REST to expose repository resources.
spring.data.rest.default-page-size= # Default size of pages.
spring.data.rest.enable-enum-translation= # Enable enum value translation via the Spring Data REST
 default resource bundle.
spring.data.rest.limit-param-name= # Name of the URL query string parameter that indicates how many
 results to return at once.
spring.data.rest.max-page-size= # Maximum size of pages.
spring.data.rest.page-param-name= # Name of the URL query string parameter that indicates what page to
 return.
spring.data.rest.return-body-on-create= # Return a response body after creating an entity.
spring.data.rest.return-body-on-update= # Return a response body after updating an entity.
spring.data.rest.sort-param-name= # Name of the URL query string parameter that indicates what direction
 to sort results.

# SOLR (SolrProperties)
spring.data.solr.host=http://127.0.0.1:8983/solr # Solr host. Ignored if "zk-host" is set.
spring.data.solr.repositories.enabled=true # Enable Solr repositories.
spring.data.solr.zk-host= # ZooKeeper host address in the form HOST:PORT.
```

```
# DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.continue-on-error=false # Do not stop if an error occurs while initializing the
 database.
spring.datasource.data= # Data (DML) script resource reference.
spring.datasource.dbcp.*= # Commons DBCP specific settings
spring.datasource.dbcp2.*= # Commons DBCP2 specific settings
spring.datasource.driver-class-name= # Fully qualified name of the JDBC driver. Auto-detected based on
 the URL by default.
spring.datasource.hikari.*= # Hikari specific settings
spring.datasource.initialize=true # Populate the database using 'data.sql'.
spring.datasource.jmx-enabled=false # Enable JMX support (if provided by the underlying pool).
spring.datasource.jndi-name= # JNDI location of the datasource. Class, url, username & password are
 ignored when set.
spring.datasource.name=testdb # Name of the datasource.
spring.datasource.password= # Login password of the database.
spring.datasource.platform=all # Platform to use in the schema resource (schema-${platform}.sql).
spring.datasource.schema= # Schema (DDL) script resource reference.
spring.datasource.separator=; # Statement separator in SQL initialization scripts.
spring.datasource.sql-script-encoding= # SQL scripts encoding.
spring.datasource.tomcat.*= # Tomcat datasource specific settings
spring.datasource.type= # Fully qualified name of the connection pool implementation to use. By default,
 it is auto-detected from the classpath.
spring.datasource.url= # JDBC url of the database.
spring.datasource.username=

# H2 Web Console (H2ConsoleProperties)
spring.h2.console.enabled=false # Enable the console.
spring.h2.console.path=/h2-console # Path at which the console will be available.
spring.h2.console.settings.trace=false # Enable trace output.
spring.h2.console.settings.web-allow-others=false # Enable remote access.

# JOOQ (JooqAutoConfiguration)
spring.jooq.sql-dialect= # SQLDialect JOOQ used when communicating with the configured datasource. For
 instance `POSTGRES`

# JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.data.jpa.repositories.enabled=true # Enable JPA repositories.
spring.jpa.database= # Target database to operate on, auto-detected by default. Can be alternatively set
 using the "databasePlatform" property.
spring.jpa.database-platform= # Name of the target database to operate on, auto-detected by default. Can
 be alternatively set using the "Database" enum.
spring.jpa.generate-ddl=false # Initialize the schema on startup.
spring.jpa.hibernate.ddl-auto= # DDL mode. This is actually a shortcut for the "hibernate.hbm2ddl.auto"
 property. Default to "create-drop" when using an embedded database, "none" otherwise.
spring.jpa.hibernate.naming.implicit-strategy= # Hibernate 5 implicit naming strategy fully qualified
 name.
spring.jpa.hibernate.naming.physical-strategy= # Hibernate 5 physical naming strategy fully qualified
 name.
spring.jpa.hibernate.naming.strategy= # Hibernate 4 naming strategy fully qualified name. Not supported
 with Hibernate 5.
spring.jpa.hibernate.use-new-id-generator-mappings= # Use Hibernate's newer IdentifierGenerator for
 AUTO, TABLE and SEQUENCE.
spring.jpa.open-in-view=true # Register OpenEntityManagerInViewInterceptor. Binds a JPA EntityManager to
 the thread for the entire processing of the request.
spring.jpa.properties.*= # Additional native properties to set on the JPA provider.
spring.jpa.show-sql=false # Enable logging of SQL statements.

# JTA (JtaAutoConfiguration)
spring.jta.enabled=true # Enable JTA support.
spring.jta.log-dir= # Transaction logs directory.
spring.jta.transaction-manager-id= # Transaction manager unique identifier.

# ATOMIKOS (AtomikosProperties)
spring.jta.atomikos.connectionfactory.borrow-connection-timeout=30 # Timeout, in seconds, for borrowing
 connections from the pool.
spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag=true # Whether or not to ignore the
 transacted flag when creating session.
spring.jta.atomikos.connectionfactory.local-transaction-mode=false # Whether or not local transactions
 are desired.
```

```
spring.jta.atomikos.connectionfactory.maintenance-interval=60 # The time, in seconds, between runs of
 the pool's maintenance thread.
spring.jta.atomikos.connectionfactory.max-idle-time=60 # The time, in seconds, after which connections
 are cleaned up from the pool.
spring.jta.atomikos.connectionfactory.max-lifetime=0 # The time, in seconds, that a connection can be
 pooled for before being destroyed. 0 denotes no limit.
spring.jta.atomikos.connectionfactory.max-pool-size=1 # The maximum size of the pool.
spring.jta.atomikos.connectionfactory.min-pool-size=1 # The minimum size of the pool.
spring.jta.atomikos.connectionfactory.reap-timeout=0 # The reap timeout, in seconds, for borrowed
 connections. 0 denotes no limit.
spring.jta.atomikos.connectionfactory.unique-resource-name=jmsConnectionFactory # The unique name used
 to identify the resource during recovery.
spring.jta.atomikos.datasource.borrow-connection-timeout=30 # Timeout, in seconds, for borrowing
 connections from the pool.
spring.jta.atomikos.datasource.default-isolation-level= # Default isolation level of connections
 provided by the pool.
spring.jta.atomikos.datasource.login-timeout= # Timeout, in seconds, for establishing a database
 connection.
spring.jta.atomikos.datasource.maintenance-interval=60 # The time, in seconds, between runs of the
 pool's maintenance thread.
spring.jta.atomikos.datasource.max-idle-time=60 # The time, in seconds, after which connections are
 cleaned up from the pool.
spring.jta.atomikos.datasource.max-lifetime=0 # The time, in seconds, that a connection can be pooled
 for before being destroyed. 0 denotes no limit.
spring.jta.atomikos.datasource.max-pool-size=1 # The maximum size of the pool.
spring.jta.atomikos.datasource.min-pool-size=1 # The minimum size of the pool.
spring.jta.atomikos.datasource.reap-timeout=0 # The reap timeout, in seconds, for borrowed connections.
 0 denotes no limit.
spring.jta.atomikos.datasource.test-query= # SQL query or statement used to validate a connection before
 returning it.
spring.jta.atomikos.datasource.unique-resource-name=dataSource # The unique name used to identify the
 resource during recovery.
spring.jta.atomikos.properties.checkpoint-interval=500 # Interval between checkpoints.
spring.jta.atomikos.properties.console-file-count=1 # Number of debug logs files that can be created.
spring.jta.atomikos.properties.console-file-limit=-1 # How many bytes can be stored at most in debug
 logs files.
spring.jta.atomikos.properties.console-file-name=tm.out # Debug logs file name.
spring.jta.atomikos.properties.console-log-level= # Console log level.
spring.jta.atomikos.properties.default-jta-timeout=10000 # Default timeout for JTA transactions.
spring.jta.atomikos.properties.enable-logging=true # Enable disk logging.
spring.jta.atomikos.properties.force-shutdown-on-vm-exit=false # Specify if a VM shutdown should trigger
 forced shutdown of the transaction core.
spring.jta.atomikos.properties.log-base-dir= # Directory in which the log files should be stored.
spring.jta.atomikos.properties.log-base-name=tmlog # Transactions log file base name.
spring.jta.atomikos.properties.max-actives=50 # Maximum number of active transactions.
spring.jta.atomikos.properties.max-timeout=300000 # Maximum timeout (in milliseconds) that can be
 allowed for transactions.
spring.jta.atomikos.properties.output-dir= # Directory in which to store the debug log files.
spring.jta.atomikos.properties.serial-jta-transactions=true # Specify if sub-transactions should be
 joined when possible.
spring.jta.atomikos.properties.service= # Transaction manager implementation that should be started.
spring.jta.atomikos.properties.threaded-two-phase-commit=true # Use different (and concurrent) threads
 for two-phase commit on the participating resources.
spring.jta.atomikos.properties.transaction-manager-unique-name= # Transaction manager's unique name.

# BITRONIX
spring.jta.bitronix.connectionfactory.acquire-increment=1 # Number of connections to create when growing
 the pool.
spring.jta.bitronix.connectionfactory.acquisition-interval=1 # Time, in seconds, to wait before trying
 to acquire a connection again after an invalid connection was acquired.
spring.jta.bitronix.connectionfactory.acquisition-timeout=30 # Timeout, in seconds, for acquiring
 connections from the pool.
spring.jta.bitronix.connectionfactory.allow-local-transactions=true # Whether or not the transaction
 manager should allow mixing XA and non-XA transactions.
spring.jta.bitronix.connectionfactory.apply-transaction-timeout=false # Whether or not the transaction
 timeout should be set on the XAResource when it is enlisted.
spring.jta.bitronix.connectionfactory.automatic-enlisting-enabled=true # Whether or not resources should
 be enlisted and delisted automatically.
spring.jta.bitronix.connectionfactory.cache-producers-consumers=true # Whether or not produces and
 consumers should be cached.
```

```
spring.jta.bitronix.connectionfactory.defer-connection-release=true # Whether or not the provider can
 run many transactions on the same connection and supports transaction interleaving.
spring.jta.bitronix.connectionfactory.ignore-recovery-failures=false # Whether or not recovery failures
 should be ignored.
spring.jta.bitronix.connectionfactory.max-idle-time=60 # The time, in seconds, after which connections
 are cleaned up from the pool.
spring.jta.bitronix.connectionfactory.max-pool-size=10 # The maximum size of the pool. 0 denotes no
 limit.
spring.jta.bitronix.connectionfactory.min-pool-size=0 # The minimum size of the pool.
spring.jta.bitronix.connectionfactory.password= # The password to use to connect to the JMS provider.
spring.jta.bitronix.connectionfactory.share-transaction-connections=false #  Whether or not connections
 in the ACCESSIBLE state can be shared within the context of a transaction.
spring.jta.bitronix.connectionfactory.test-connections=true # Whether or not connections should be
 tested when acquired from the pool.
spring.jta.bitronix.connectionfactory.two-pc-ordering-position=1 # The position that this
 resource should take during two-phase commit (always first is Integer.MIN_VALUE, always last is
 Integer.MAX_VALUE).
spring.jta.bitronix.connectionfactory.unique-name=jmsConnectionFactory # The unique name used to
 identify the resource during recovery.
spring.jta.bitronix.connectionfactory.use-tm-join=true Whether or not TMJOIN should be used when
 starting XAResources.
spring.jta.bitronix.connectionfactory.user= # The user to use to connect to the JMS provider.
spring.jta.bitronix.datasource.acquire-increment=1 # Number of connections to create when growing the
 pool.
spring.jta.bitronix.datasource.acquisition-interval=1 # Time, in seconds, to wait before trying to
 acquire a connection again after an invalid connection was acquired.
spring.jta.bitronix.datasource.acquisition-timeout=30 # Timeout, in seconds, for acquiring connections
 from the pool.
spring.jta.bitronix.datasource.allow-local-transactions=true # Whether or not the transaction manager
 should allow mixing XA and non-XA transactions.
spring.jta.bitronix.datasource.apply-transaction-timeout=false # Whether or not the transaction timeout
 should be set on the XAResource when it is enlisted.
spring.jta.bitronix.datasource.automatic-enlisting-enabled=true # Whether or not resources should be
 enlisted and delisted automatically.
spring.jta.bitronix.datasource.cursor-holdability= # The default cursor holdability for connections.
spring.jta.bitronix.datasource.defer-connection-release=true # Whether or not the database can run many
 transactions on the same connection and supports transaction interleaving.
spring.jta.bitronix.datasource.enable-jdbc4-connection-test= # Whether or not Connection.isValid() is
 called when acquiring a connection from the pool.
spring.jta.bitronix.datasource.ignore-recovery-failures=false # Whether or not recovery failures should
 be ignored.
spring.jta.bitronix.datasource.isolation-level= # The default isolation level for connections.
spring.jta.bitronix.datasource.local-auto-commit= # The default auto-commit mode for local transactions.
spring.jta.bitronix.datasource.login-timeout= # Timeout, in seconds, for establishing a database
 connection.
spring.jta.bitronix.datasource.max-idle-time=60 # The time, in seconds, after which connections are
 cleaned up from the pool.
spring.jta.bitronix.datasource.max-pool-size=10 # The maximum size of the pool. 0 denotes no limit.
spring.jta.bitronix.datasource.min-pool-size=0 # The minimum size of the pool.
spring.jta.bitronix.datasource.prepared-statement-cache-size=0 # The target size of the prepared
 statement cache. 0 disables the cache.
spring.jta.bitronix.datasource.share-transaction-connections=false #  Whether or not connections in the
 ACCESSIBLE state can be shared within the context of a transaction.
spring.jta.bitronix.datasource.test-query= # SQL query or statement used to validate a connection before
 returning it.
spring.jta.bitronix.datasource.two-pc-ordering-position=1 # The position that this resource should take
 during two-phase commit (always first is Integer.MIN_VALUE, always last is Integer.MAX_VALUE).
spring.jta.bitronix.datasource.unique-name=dataSource # The unique name used to identify the resource
 during recovery.
spring.jta.bitronix.datasource.use-tm-join=true Whether or not TMJOIN should be used when starting
 XAResources.
spring.jta.bitronix.properties.allow-multiple-lrc=false # Allow multiple LRC resources to be enlisted
 into the same transaction.
spring.jta.bitronix.properties.asynchronous2-pc=false # Enable asynchronously execution of two phase
 commit.
spring.jta.bitronix.properties.background-recovery-interval-seconds=60 # Interval in seconds at which to
 run the recovery process in the background.
spring.jta.bitronix.properties.current-node-only-recovery=true # Recover only the current node.
spring.jta.bitronix.properties.debug-zero-resource-transaction=false # Log the creation and commit call
 stacks of transactions executed without a single enlisted resource.
spring.jta.bitronix.properties.default-transaction-timeout=60 # Default transaction timeout in seconds.
spring.jta.bitronix.properties.disable-jmx=false # Enable JMX support.
spring.jta.bitronix.properties.exception-analyzer= # Set the fully qualified name of the exception
 analyzer implementation to use.
spring.jta.bitronix.properties.filter-log-status=false # Enable filtering of logs so that only mandatory
 logs are written.
spring.jta.bitronix.properties.force-batching-enabled=true #  Set if disk forces are batched.
spring.jta.bitronix.properties.forced-write-enabled=true # Set if logs are forced to disk.
spring.jta.bitronix.properties.graceful-shutdown-interval=60 # Maximum amount of seconds the TM will
 wait for transactions to get done before aborting them at shutdown time.
spring.jta.bitronix.properties.jndi-transaction-synchronization-registry-name= # JNDI name of the
 TransactionSynchronizationRegistry.
spring.jta.bitronix.properties.jndi-user-transaction-name= # JNDI name of the UserTransaction.
spring.jta.bitronix.properties.journal=disk # Name of the journal. Can be 'disk', 'null' or a class
 name.
spring.jta.bitronix.properties.log-part1-filename=btm1.tlog # Name of the first fragment of the journal.
spring.jta.bitronix.properties.log-part2-filename=btm2.tlog # Name of the second fragment of the
 journal.
spring.jta.bitronix.properties.max-log-size-in-mb=2 # Maximum size in megabytes of the journal
 fragments.
spring.jta.bitronix.properties.resource-configuration-filename= # ResourceLoader configuration file
 name.
spring.jta.bitronix.properties.server-id= # ASCII ID that must uniquely identify this TM instance.
 Default to the machine's IP address.
spring.jta.bitronix.properties.skip-corrupted-logs=false # Skip corrupted transactions log entries.
spring.jta.bitronix.properties.warn-about-zero-resource-transaction=true # Log a warning for
 transactions executed without a single enlisted resource.

# NARAYANA (NarayanaProperties)
spring.jta.narayana.default-timeout=60 # Transaction timeout in seconds.
spring.jta.narayana.expiry-
scanners=com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner # Comma-
separated list of expiry scanners.
spring.jta.narayana.log-dir= # Transaction object store directory.
spring.jta.narayana.one-phase-commit=true # Enable one phase commit optimisation.
spring.jta.narayana.periodic-recovery-period=120 # Interval in which periodic recovery scans are
 performed in seconds.
spring.jta.narayana.recovery-backoff-period=10 # Back off period between first and second phases of the
 recovery scan in seconds.
spring.jta.narayana.recovery-db-pass= # Database password to be used by recovery manager.
spring.jta.narayana.recovery-db-user= # Database username to be used by recovery manager.
spring.jta.narayana.recovery-jms-pass= # JMS password to be used by recovery manager.
spring.jta.narayana.recovery-jms-user= # JMS username to be used by recovery manager.
spring.jta.narayana.recovery-modules= # Comma-separated list of recovery modules.
spring.jta.narayana.transaction-manager-id=1 # Unique transaction manager id.
spring.jta.narayana.xa-resource-orphan-filters= # Comma-separated list of of orphan filters.

# EMBEDDED MONGODB (EmbeddedMongoProperties)
spring.mongodb.embedded.features=SYNC_DELAY # Comma-separated list of features to enable.
spring.mongodb.embedded.storage.databaseDir= # Directory used for data storage.
spring.mongodb.embedded.storage.oplogSize= # Maximum size of the oplog in megabytes.
spring.mongodb.embedded.storage.replSetName= # Name of the replica set.
spring.mongodb.embedded.version=2.6.10 # Version of Mongo to use.

# REDIS (RedisProperties)
spring.redis.cluster.max-redirects= # Maximum number of redirects to follow when executing commands
 across the cluster.
spring.redis.cluster.nodes= # Comma-separated list of "host:port" pairs to bootstrap from.
spring.redis.database=0 # Database index used by the connection factory.
spring.redis.host=localhost # Redis server host.
spring.redis.password= # Login password of the redis server.
spring.redis.pool.max-active=8 # Max number of connections that can be allocated by the pool at a given
 time. Use a negative value for no limit.
spring.redis.pool.max-idle=8 # Max number of "idle" connections in the pool. Use a negative value to
 indicate an unlimited number of idle connections.
spring.redis.pool.max-wait=-1 # Maximum amount of time (in milliseconds) a connection allocation
 should block before throwing an exception when the pool is exhausted. Use a negative value to block
 indefinitely.
spring.redis.pool.min-idle=0 # Target for the minimum number of idle connections to maintain in the
 pool. This setting only has an effect if it is positive.
spring.redis.port=6379 # Redis server port.
spring.redis.sentinel.master= # Name of Redis server.
spring.redis.sentinel.nodes= # Comma-separated list of host:port pairs.
spring.redis.timeout=0 # Connection timeout in milliseconds.


# ----------------------------------------
# INTEGRATION PROPERTIES
# ----------------------------------------

# ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url= # URL of the ActiveMQ broker. Auto-generated by default. For instance
 `tcp://localhost:61616`
spring.activemq.in-memory=true # Specify if the default broker URL should be in memory. Ignored if an
 explicit broker has been specified.
spring.activemq.password= # Login password of the broker.
spring.activemq.user= # Login user of the broker.
spring.activemq.packages.trust-all=false # Trust all packages
spring.activemq.packages.trusted= # Comma-separated list of specific packages to trust (when not
 trusting all packages).
spring.activemq.pool.configuration.*= # See PooledConnectionFactory
spring.activemq.pool.enabled=false # Whether a PooledConnectionFactory should be created instead of a
 regular ConnectionFactory.
spring.activemq.pool.expiry-timeout=0 # Connection expiration timeout in milliseconds.
spring.activemq.pool.idle-timeout=30000 # Connection idle timeout in milliseconds.
spring.activemq.pool.max-connections=1 # Maximum number of pooled connections.

# ARTEMIS (ArtemisProperties)
spring.artemis.embedded.cluster-password= # Cluster password. Randomly generated on startup by default.
spring.artemis.embedded.data-directory= # Journal file directory. Not necessary if persistence is turned
 off.
spring.artemis.embedded.enabled=true # Enable embedded mode if the Artemis server APIs are available.
spring.artemis.embedded.persistent=false # Enable persistent store.
spring.artemis.embedded.queues= # Comma-separated list of queues to create on startup.
spring.artemis.embedded.server-id= # Server id. By default, an auto-incremented counter is used.
spring.artemis.embedded.topics= # Comma-separated list of topics to create on startup.
spring.artemis.host=localhost # Artemis broker host.
spring.artemis.mode= # Artemis deployment mode, auto-detected by default. Can be explicitly set to
 "native" or "embedded".
spring.artemis.port=61616 # Artemis broker port.

# SPRING BATCH (BatchProperties)
spring.batch.initializer.enabled=true # Create the required batch tables on startup if necessary.
spring.batch.job.enabled=true # Execute all Spring Batch jobs in the context on startup.
spring.batch.job.names= # Comma-separated list of job names to execute on startup (For instance
 `job1,job2`). By default, all Jobs found in the context are executed.
spring.batch.schema=classpath:org/springframework/batch/core/schema-@@platform@@.sql # Path to the SQL
 file to use to initialize the database schema.
spring.batch.table-prefix= # Table prefix for all the batch meta-data tables.

# HORNETQ (HornetQProperties)
spring.hornetq.embedded.cluster-password= # Cluster password. Randomly generated on startup by default.
spring.hornetq.embedded.data-directory= # Journal file directory. Not necessary if persistence is turned
 off.
spring.hornetq.embedded.enabled=true # Enable embedded mode if the HornetQ server APIs are available.
spring.hornetq.embedded.persistent=false # Enable persistent store.
spring.hornetq.embedded.queues= # Comma-separated list of queues to create on startup.
spring.hornetq.embedded.server-id= # Server id. By default, an auto-incremented counter is used.
spring.hornetq.embedded.topics= # Comma-separated list of topics to create on startup.
spring.hornetq.host=localhost # HornetQ broker host.
spring.hornetq.mode= # HornetQ deployment mode, auto-detected by default. Can be explicitly set to
 "native" or "embedded".
spring.hornetq.port=5445 # HornetQ broker port.

# JMS (JmsProperties)
spring.jms.jndi-name= # Connection factory JNDI name. When set, takes precedence to others connection
 factory auto-configurations.
spring.jms.listener.acknowledge-mode= # Acknowledge mode of the container. By default, the listener is
 transacted with automatic acknowledgment.
spring.jms.listener.auto-startup=true # Start the container automatically on startup.
spring.jms.listener.concurrency= # Minimum number of concurrent consumers.
spring.jms.listener.max-concurrency= # Maximum number of concurrent consumers.
spring.jms.pub-sub-domain=false # Specify if the default destination type is topic.
```

```
# RABBIT (RabbitProperties)
spring.rabbitmq.addresses= # Comma-separated list of addresses to which the client should connect.
spring.rabbitmq.cache.channel.checkout-timeout= # Number of milliseconds to wait to obtain a channel if
 the cache size has been reached.
spring.rabbitmq.cache.channel.size= # Number of channels to retain in the cache.
spring.rabbitmq.cache.connection.mode=CHANNEL # Connection factory cache mode.
spring.rabbitmq.cache.connection.size= # Number of connections to cache.
spring.rabbitmq.dynamic=true # Create an AmqpAdmin bean.
spring.rabbitmq.host=localhost # RabbitMQ host.
spring.rabbitmq.listener.acknowledge-mode= # Acknowledge mode of container.
spring.rabbitmq.listener.auto-startup=true # Start the container automatically on startup.
spring.rabbitmq.listener.concurrency= # Minimum number of consumers.
spring.rabbitmq.listener.default-requeue-rejected= # Whether or not to requeue delivery failures;
 default `true`.
spring.rabbitmq.listener.max-concurrency= # Maximum number of consumers.
spring.rabbitmq.listener.prefetch= # Number of messages to be handled in a single request. It should be
 greater than or equal to the transaction size (if used).
spring.rabbitmq.listener.retry.enabled=false # Whether or not publishing retries are enabled.
spring.rabbitmq.listener.retry.initial-interval=1000 # Interval between the first and second attempt to
 deliver a message.
spring.rabbitmq.listener.retry.max-attempts=3 # Maximum number of attempts to deliver a message.
spring.rabbitmq.listener.retry.max-interval=10000 # Maximum number of attempts to deliver a message.
spring.rabbitmq.listener.retry.multiplier=1.0 # A multiplier to apply to the previous delivery retry
 interval.
spring.rabbitmq.listener.retry.stateless=true # Whether or not retry is stateless or stateful.
spring.rabbitmq.listener.transaction-size= # Number of messages to be processed in a transaction. For
 best results it should be less than or equal to the prefetch count.
spring.rabbitmq.password= # Login to authenticate against the broker.
spring.rabbitmq.port=5672 # RabbitMQ port.
spring.rabbitmq.publisher-confirms=false # Enable publisher confirms.
spring.rabbitmq.publisher-returns=false # Enable publisher returns.
spring.rabbitmq.requested-heartbeat= # Requested heartbeat timeout, in seconds; zero for none.
spring.rabbitmq.ssl.enabled=false # Enable SSL support.
spring.rabbitmq.ssl.key-store= # Path to the key store that holds the SSL certificate.
spring.rabbitmq.ssl.key-store-password= # Password used to access the key store.
spring.rabbitmq.ssl.trust-store= # Trust store that holds SSL certificates.
spring.rabbitmq.ssl.trust-store-password= # Password used to access the trust store.
spring.rabbitmq.template.mandatory=false # Enable mandatory messages.
spring.rabbitmq.template.receive-timeout=0 # Timeout for `receive()` methods.
spring.rabbitmq.template.reply-timeout=5000 # Timeout for `sendAndReceive()` methods.
spring.rabbitmq.template.retry.enabled=false # Set to true to enable retries in the `RabbitTemplate`.
spring.rabbitmq.template.retry.initial-interval=1000 # Interval between the first and second attempt to
 publish a message.
spring.rabbitmq.template.retry.max-attempts=3 # Maximum number of attempts to publish a message.
spring.rabbitmq.template.retry.max-interval=10000 # Maximum number of attempts to publish a message.
spring.rabbitmq.template.retry.multiplier=1.0 # A multiplier to apply to the previous publishing retry
 interval.
spring.rabbitmq.username= # Login user to authenticate to the broker.
spring.rabbitmq.virtual-host= # Virtual host to use when connecting to the broker.


# ----------------------------------------
# ACTUATOR PROPERTIES
# ----------------------------------------

# ENDPOINTS (AbstractEndpoint subclasses)
endpoints.enabled=true # Enable endpoints.
endpoints.sensitive= # Default endpoint sensitive setting.
endpoints.actuator.enabled=true # Enable the endpoint.
endpoints.actuator.path= # Endpoint URL path.
endpoints.actuator.sensitive=false # Enable security on the endpoint.
endpoints.autoconfig.enabled= # Enable the endpoint.
endpoints.autoconfig.id= # Endpoint identifier.
endpoints.autoconfig.path= # Endpoint path.
endpoints.autoconfig.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.beans.enabled= # Enable the endpoint.
endpoints.beans.id= # Endpoint identifier.
endpoints.beans.path= # Endpoint path.
endpoints.beans.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.configprops.enabled= # Enable the endpoint.
endpoints.configprops.id= # Endpoint identifier.
endpoints.configprops.keys-to-sanitize=password,secret,key,token,.*credentials.*,vcap_services # Keys
 that should be sanitized. Keys can be simple strings that the property ends with or regex expressions.
endpoints.configprops.path= # Endpoint path.
endpoints.configprops.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.docs.curies.enabled=false # Enable the curie generation.
endpoints.docs.enabled=true # Enable actuator docs endpoint.
endpoints.docs.path=/docs #
endpoints.docs.sensitive=false #
endpoints.dump.enabled= # Enable the endpoint.
endpoints.dump.id= # Endpoint identifier.
endpoints.dump.path= # Endpoint path.
endpoints.dump.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.env.enabled= # Enable the endpoint.
endpoints.env.id= # Endpoint identifier.
endpoints.env.keys-to-sanitize=password,secret,key,token,.*credentials.*,vcap_services # Keys that
 should be sanitized. Keys can be simple strings that the property ends with or regex expressions.
endpoints.env.path= # Endpoint path.
endpoints.env.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.flyway.enabled= # Enable the endpoint.
endpoints.flyway.id= # Endpoint identifier.
endpoints.flyway.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.health.enabled= # Enable the endpoint.
endpoints.health.id= # Endpoint identifier.
endpoints.health.mapping.*= # Mapping of health statuses to HttpStatus codes. By default, registered
 health statuses map to sensible defaults (i.e. UP maps to 200).
endpoints.health.path= # Endpoint path.
endpoints.health.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.health.time-to-live=1000 # Time to live for cached result, in milliseconds.
endpoints.info.enabled= # Enable the endpoint.
endpoints.info.id= # Endpoint identifier.
endpoints.info.path= # Endpoint path.
endpoints.info.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.jolokia.enabled=true # Enable Jolokia endpoint.
endpoints.jolokia.path=/jolokia # Endpoint URL path.
endpoints.jolokia.sensitive=true # Enable security on the endpoint.
endpoints.liquibase.enabled= # Enable the endpoint.
endpoints.liquibase.id= # Endpoint identifier.
endpoints.liquibase.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.logfile.enabled=true # Enable the endpoint.
endpoints.logfile.path=/logfile # Endpoint URL path.
endpoints.logfile.sensitive=true # Enable security on the endpoint.
endpoints.mappings.enabled= # Enable the endpoint.
endpoints.mappings.id= # Endpoint identifier.
endpoints.mappings.path= # Endpoint path.
endpoints.mappings.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.metrics.enabled= # Enable the endpoint.
endpoints.metrics.filter.enabled=true # Enable the metrics servlet filter.
endpoints.metrics.filter.gauge-submissions=merged # Http filter gauge submissions (merged, per-http-
method)
endpoints.metrics.filter.counter-submissions=merged # Http filter counter submissions (merged, per-http-
method)
endpoints.metrics.id= # Endpoint identifier.
endpoints.metrics.path= # Endpoint path.
endpoints.metrics.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.shutdown.enabled= # Enable the endpoint.
endpoints.shutdown.id= # Endpoint identifier.
endpoints.shutdown.path= # Endpoint path.
endpoints.shutdown.sensitive= # Mark if the endpoint exposes sensitive information.
endpoints.trace.enabled= # Enable the endpoint.
endpoints.trace.id= # Endpoint identifier.
endpoints.trace.path= # Endpoint path.
endpoints.trace.sensitive= # Mark if the endpoint exposes sensitive information.

# ENDPOINTS CORS CONFIGURATION (EndpointCorsProperties)
endpoints.cors.allow-credentials= # Set whether credentials are supported. When not set, credentials are
 not supported.
endpoints.cors.allowed-headers= # Comma-separated list of headers to allow in a request. '*' allows all
 headers.
endpoints.cors.allowed-methods=GET # Comma-separated list of methods to allow. '*' allows all methods.
endpoints.cors.allowed-origins= # Comma-separated list of origins to allow. '*' allows all origins. When
 not set, CORS support is disabled.
endpoints.cors.exposed-headers= # Comma-separated list of headers to include in a response.
endpoints.cors.max-age=1800 # How long, in seconds, the response from a pre-flight request can be cached
 by clients.

# JMX ENDPOINT (EndpointMBeanExportProperties)
endpoints.jmx.domain= # JMX domain name. Initialized with the value of 'spring.jmx.default-domain' if
 set.
endpoints.jmx.enabled=true # Enable JMX export of all endpoints.
endpoints.jmx.static-names= # Additional static properties to append to all ObjectNames of MBeans
 representing Endpoints.
endpoints.jmx.unique-names=false # Ensure that ObjectNames are modified in case of conflict.

# JOLOKIA (JolokiaProperties)
jolokia.config.*= # See Jolokia manual

# MANAGEMENT HTTP SERVER (ManagementServerProperties)
management.add-application-context-header=true # Add the "X-Application-Context" HTTP header in each
 response.
management.address= # Network address that the management endpoints should bind to.
management.context-path= # Management endpoint context-path. For instance `/actuator`
management.port= # Management endpoint HTTP port. Use the same port as the application by default.
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).

# HEALTH INDICATORS (previously health.*)
management.health.db.enabled=true # Enable database health check.
management.health.defaults.enabled=true # Enable default health indicators.
management.health.diskspace.enabled=true # Enable disk space health check.
management.health.diskspace.path= # Path used to compute the available disk space.
management.health.diskspace.threshold=0 # Minimum disk space that should be available, in bytes.
management.health.elasticsearch.enabled=true # Enable elasticsearch health check.
management.health.elasticsearch.indices= # Comma-separated index names.
management.health.elasticsearch.response-timeout=100 # The time, in milliseconds, to wait for a response
 from the cluster.
management.health.jms.enabled=true # Enable JMS health check.
management.health.mail.enabled=true # Enable Mail health check.
management.health.mongo.enabled=true # Enable MongoDB health check.
management.health.rabbit.enabled=true # Enable RabbitMQ health check.
management.health.redis.enabled=true # Enable Redis health check.
management.health.solr.enabled=true # Enable Solr health check.
management.health.status.order=DOWN, OUT_OF_SERVICE, UNKNOWN, UP # Comma-separated list of health
 statuses in order of severity.

# INFO CONTRIBUTORS (InfoContributorProperties)
management.info.build.enabled=true # Enable build info.
management.info.build.mode=simple # Mode to use to expose build information.
management.info.defaults.enabled=true # Enable default info contributors.
management.info.env.enabled=true # Enable environment info.
management.info.git.enabled=true # Enable git info.
management.info.git.mode=simple # Mode to use to expose git information.

# TRACING (TraceProperties)
management.trace.include=request-headers,response-headers,errors # Items to be included in the trace.

# REMOTE SHELL
shell.auth=simple # Authentication type. Auto-detected according to the environment.
shell.auth.jaas.domain=my-domain # JAAS domain.
shell.auth.key.path= # Path to the authentication key. This should point to a valid ".pem" file.
shell.auth.simple.user.name=user # Login user.
shell.auth.simple.user.password= # Login password.
shell.auth.spring.roles=ADMIN # Comma-separated list of required roles to login to the CRaSH console.
shell.command-path-patterns=classpath*:/commands/**,classpath*:/crash/commands/** # Patterns to use to
 look for commands.
shell.command-refresh-interval=-1 # Scan for changes and update the command if necessary (in seconds).
shell.config-path-patterns=classpath*:/crash/* # Patterns to use to look for configurations.
shell.disabled-commands=jpa*,jdbc*,jndi* # Comma-separated list of commands to disable.
shell.disabled-plugins= # Comma-separated list of plugins to disable. Certain plugins are disabled by
 default based on the environment.
shell.ssh.auth-timeout = # Number of milliseconds after user will be prompted to login again.
shell.ssh.enabled=true # Enable CRaSH SSH support.
shell.ssh.idle-timeout = # Number of milliseconds after which unused connections are closed.
shell.ssh.key-path= # Path to the SSH server key.
shell.ssh.port=2000 # SSH port.
shell.telnet.enabled=false # Enable CRaSH telnet support. Enabled by default if the TelnetPlugin is
 available.
shell.telnet.port=5000 # Telnet port.

# METRICS EXPORT (MetricExportProperties)
spring.metrics.export.aggregate.key-pattern= # Pattern that tells the aggregator what to do with the
 keys from the source repository.
spring.metrics.export.aggregate.prefix= # Prefix for global repository if active.
spring.metrics.export.delay-millis=5000 # Delay in milliseconds between export ticks. Metrics are
 exported to external sources on a schedule with this delay.
spring.metrics.export.enabled=true # Flag to enable metric export (assuming a MetricWriter is
 available).
spring.metrics.export.excludes= # List of patterns for metric names to exclude. Applied after the
 includes.
spring.metrics.export.includes= # List of patterns for metric names to include.
spring.metrics.export.redis.key=keys.spring.metrics # Key for redis repository export (if active).
spring.metrics.export.redis.prefix=spring.metrics # Prefix for redis repository if active.
spring.metrics.export.send-latest= # Flag to switch off any available optimizations based on not
 exporting unchanged metric values.
spring.metrics.export.statsd.host= # Host of a statsd server to receive exported metrics.
spring.metrics.export.statsd.port=8125 # Port of a statsd server to receive exported metrics.
spring.metrics.export.statsd.prefix= # Prefix for statsd exported metrics.
spring.metrics.export.triggers.*= # Specific trigger properties per MetricWriter bean name.


# ----------------------------------------
# DEVTOOLS PROPERTIES
# ----------------------------------------

# DEVTOOLS (DevToolsProperties)
spring.devtools.livereload.enabled=true # Enable a livereload.com compatible server.
spring.devtools.livereload.port=35729 # Server port.
spring.devtools.restart.additional-exclude= # Additional patterns that should be excluded from
 triggering a full restart.
spring.devtools.restart.additional-paths= # Additional paths to watch for changes.
spring.devtools.restart.enabled=true # Enable automatic restart.
spring.devtools.restart.exclude=META-INF/maven/**,META-INF/resources/**,resources/**,static/**,public/
**,templates/**,**/*Test.class,**/*Tests.class,git.properties # Patterns that should be excluded from
 triggering a full restart.
spring.devtools.restart.poll-interval=1000 # Amount of time (in milliseconds) to wait between polling
 for classpath changes.
spring.devtools.restart.quiet-period=400 # Amount of quiet time (in milliseconds) required without any
 classpath changes before a restart is triggered.
spring.devtools.restart.trigger-file= # Name of a specific file that when changed will trigger the
 restart check. If not specified any classpath file change will trigger the restart.

# REMOTE DEVTOOLS (RemoteDevToolsProperties)
spring.devtools.remote.context-path=/.~~spring-boot!~ # Context path used to handle the remote
 connection.
spring.devtools.remote.debug.enabled=true # Enable remote debug support.
spring.devtools.remote.debug.local-port=8000 # Local remote debug server port.
spring.devtools.remote.proxy.host= # The host of the proxy to use to connect to the remote application.
spring.devtools.remote.proxy.port= # The port of the proxy to use to connect to the remote application.
spring.devtools.remote.restart.enabled=true # Enable remote restart.
spring.devtools.remote.secret= # A shared secret required to establish a connection (required to enable
 remote support).
spring.devtools.remote.secret-header-name=X-AUTH-TOKEN # HTTP header used to transfer the shared secret.
```

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`. However, it is possible to write part of the meta-data manually for corner cases or more advanced use cases.

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” and additional values hint categorized under "hints":

```
{"groups": [
    {
        "name": "server",
        "type": "org.springframework.boot.autoconfigure.web.ServerProperties",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
        "name": "spring.jpa.hibernate",
        "type": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate",
        "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties",
        "sourceMethod": "getHibernate()"
    }
    ...
],"properties": [
    {
        "name": "server.port",
        "type": "java.lang.Integer",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
    },
    {
        "name": "server.servlet-path",
        "type": "java.lang.String",
        "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties",
        "defaultValue": "/"
    },
    {
          "name": "spring.jpa.hibernate.ddl-auto",
          "type": "java.lang.String",
          "description": "DDL mode. This is actually a shortcut for the \"hibernate.hbm2ddl.auto\"
 property.",
          "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"
    }
    ...
],"hints": [
    {
        "name": "spring.jpa.hibernate.ddl-auto",
        "values": [
            {
                "value": "none",
                "description": "Disable DDL handling."
            },
            {
                "value": "validate",
                "description": "Validate the schema, make no changes to the database."
            },
            {
                "value": "update",
                "description": "Update the schema if necessary."
            },
            {
                "value": "create",
                "description": "Create the schema and destroy previous data."
            },
            {
                "value": "create-drop",
                "description": "Create and then destroy the schema at the end of the session."
            }
        ]
    }
]}
```

Each “property” is a configuration item that the user specifies with a given value. For example `server.port` and `server.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.

Finally, “hints” are additional information used to assist the user in configuring a given property. When configuring the `spring.jpa.hibernate.ddl-auto` property, a tool can use it to offer some auto-completion help for the `none`, `validate`, `update`, `create` and `create-drop` values.

Group Attributes

The JSON object contained in the `groups` array can contain the 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

Name	Type	Purpose
		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`. 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.
`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.
`deprecation`	Deprecation	Specify if the property is deprecated. May be omitted if the field is not deprecated or if that information is not known. See below for more details.

The JSON object contained in the `deprecation` attribute of each `properties` element can contain the following attributes:

Name	Type	Purpose
`reason`	String	A short description of the reason why the property was deprecated. May be omitted if no reason 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 (`.`).
`replacement`	String	The full name of the property that is *replacing* this deprecated property. May be omitted if there is no replacement for this property.

Note

Prior to Spring Boot 1.3, a single `deprecated` boolean attribute can be used instead of the `deprecation` element. This is still supported in a deprecated fashion and should no longer be used. If no reason and replacement are available, an empty `deprecation` object should be set.

Deprecation can also be specified declaratively in code by adding the `@DeprecatedConfigurationProperty` annotation to the getter exposing the deprecated property. For instance, let's assume the `app.foo.target` property was confusing and was renamed to `app.foo.name`

```
@ConfigurationProperties("app.foo")
public class FooProperties {

    private String name;

    public String getName() { ... }

    public void setName(String name) { ... }

    @DeprecatedConfigurationProperty(replacement = "app.foo.name")
    @Deprecated
    public String getTarget() {
        return getName();
    }

    @Deprecated
    public void setTarget(String target) {
        setName(target);
    }
}
```

The code above makes sure that the deprecated property still works (delegating to the `name` property behind the scenes). Once the `getTarget` and `setTarget` methods can be removed from your public API, the automatic deprecation hint in the meta-data will go away as well.

Hint Attributes

The JSON object contained in the `hints` array can contain the following attributes:

Name	Type	Purpose
`name`	String	The full name of the property that this hint refers to. Names are in lowercase dashed form (e.g. `server.servlet-path`). If the property refers to a map (e.g. `system.contexts`) the hint either

Name	Type	Purpose
		applies to the *keys* of the map (`system.context.keys`) or the values (`system.context.values`). This attribute is mandatory.
`values`	ValueHint[]	A list of valid values as defined by the `ValueHint` object (see below). Each entry defines the value and may have a description
`providers`	ValueProvider[]	A list of providers as defined by the `ValueProvider` object (see below). Each entry defines the name of the provider and its parameters, if any.

The JSON object contained in the `values` attribute of each `hint` element can contain the following attributes:

Name	Type	Purpose
`value`	Object	A valid value for the element to which the hint refers to. Can also be an array of value(s) if the type of the property is an array. This attribute is mandatory.
`description`	String	A short description of the value 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 (`.`).

The JSON object contained in the `providers` attribute of each `hint` element can contain the following attributes:

Name	Type	Purpose
`name`	String	The name of the provider to use to offer additional content assistance for the element to which the hint refers to.
`parameters`	JSON object	Any additional parameter that the provider supports (check the documentation of the provider for more details).

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, you could bind two separate classes to the same prefix, with each potentially offering overlap of property names. While this is not supposed to be a frequent scenario, consumers of meta-data should take care to ensure that they support such scenarios.

B.2 Providing manual hints

To improve the user experience and further assist the user in configuring a given property, you can provide additional meta-data that:

1. Describes the list of potential values for a property.
2. Associates a provider to attach a well-defined semantic to a property so that a tool can discover the list of potential values based on the project’s context.

Value hint

The `name` attribute of each hint refers to the `name` of a property. In the initial example above, we provide 5 values for the `spring.jpa.hibernate.ddl-auto` property: `none`, `validate`, `update`, `create` and `create-drop`. Each value may have a description as well.

If your property is of type `Map`, you can provide hints for both the keys and the values (but not for the map itself). The special `.keys` and `.values` suffixes must be used to refer to the keys and the values respectively.

Let’s assume a `foo.contexts` that maps magic String values to an integer:

```
@ConfigurationProperties("foo")
public class FooProperties {

    private Map<String,Integer> contexts;
    // getters and setters
}
```

The magic values are foo and bar for instance. In order to offer additional content assistance for the keys, you could add the following to the manual meta-data of the module:

```
{"hints": [
    {
        "name": "foo.contexts.keys",
        "values": [
            {
                "value": "foo"
            },
            {
                "value": "bar"
            }
        ]
    }
]}
```

Note

Of course, you should have an `Enum` for those two values instead. This is by far the most effective approach to auto-completion if your IDE supports it.

Value provider

Providers are a powerful way of attaching semantics to a property. We define in the section below the official providers that you can use for your own hints. Bare in mind however that your favorite IDE may implement some of these or none of them. It could eventually provide its own as well.

Note

As this is a new feature, IDE vendors will have to catch up with this new feature.

The table below summarizes the list of supported providers:

Name	Description
`any`	Permit any additional value to be provided.

Name	Description
`class-reference`	Auto-complete the classes available in the project. Usually constrained by a base class that is specified via the `target` parameter.
`handle-as`	Handle the property as if it was defined by the type defined via the mandatory `target` parameter.
`logger-name`	Auto-complete valid logger names. Typically, package and class names available in the current project can be auto-completed.
`spring-bean-reference`	Auto-complete the available bean names in the current project. Usually constrained by a base class that is specified via the `target` parameter.
`spring-profile-name`	Auto-complete the available Spring profile names in the project.

Tip

No more than one provider can be active for a given property but you can specify several providers if they can all manage the property *in some ways*. Make sure to place the most powerful provider first as the IDE must use the first one in the JSON section it can handle. If no provider for a given property is supported, no special content assistance is provided either.

Any

The **any** provider permits any additional values to be provided. Regular value validation based on the property type should be applied if this is supported.

This provider will be typically used if you have a list of values and any extra values are still to be considered as valid.

The example below offers on and `off` as auto-completion values for `system.state`; any other value is also allowed:

```
{"hints": [
    {
        "name": "system.state",
        "values": [
            {
                "value": "on"
            },
            {
                "value": "off"
            }
        ],
        "providers": [
            {
                "name": "any"
            }
        ]
    }
]}
```

Class reference

The **class-reference** provider auto-completes classes available in the project. This provider supports these parameters:

Parameter	Type	Default value	Description
`target`	`String` (`Class`)	*none*	The fully qualified name of the class that should be assignable to the chosen value. Typically used to filter out non candidate classes. Note that this information can be provided by the type itself by exposing a class with the appropriate upper bound.
`concrete`	`boolean`	true	Specify if only concrete classes are to be considered as valid candidates.

The meta-data snippet below corresponds to the standard `server.jsp-servlet.class-name` property that defines the `JspServlet` class name to use:

```
{"hints": [
    {
        "name": "server.jsp-servlet.class-name",
        "providers": [
            {
                "name": "class-reference",
                "parameters": {
                    "target": "javax.servlet.http.HttpServlet"
                }
            }
        ]
    }
]}
```

Handle As

The **handle-as** provider allows you to substitute the type of the property to a more high-level type. This typically happens when the property has a `java.lang.String` type because you don’t want your configuration classes to rely on classes that may not be on the classpath. This provider supports these parameters:

Parameter	Type	Default value	Description
`target`	`String` (`Class`)	*none*	The fully qualified name of the type to consider for the property. This parameter is mandatory.

The following types can be used:

- Any `java.lang.Enum` that lists the possible values for the property (By all means, try to define the property with the `Enum` type instead as no further hint should be required for the IDE to auto-complete the values).
- `java.nio.charset.Charset`: auto-completion of charset/encoding values (e.g. `UTF-8`)
- `java.util.Locale`: auto-completion of locales (e.g. `en_US`)
- `org.springframework.util.MimeType`: auto-completion of content type values (e.g. `text/plain`)
- `org.springframework.core.io.Resource`: auto-completion of Spring’s Resource abstraction to refer to a file on the filesystem or on the classpath. (e.g. `classpath:/foo.properties`)

> **Note**
>
> If multiple values can be provided, use a `Collection` or *Array* type to teach the IDE about it.

The meta-data snippet below corresponds to the standard `liquibase.change-log` property that defines the path to the changelog to use. It is actually used internally as a `org.springframework.core.io.Resource` but cannot be exposed as such as we need to keep the original String value to pass it to the Liquibase API.

```
{"hints": [
    {
        "name": "liquibase.change-log",
        "providers": [
            {
                "name": "handle-as",
                "parameters": {
                    "target": "org.springframework.core.io.Resource"
                }
            }
        ]
    }
]}
```

Logger name

The **logger-name** provider auto-completes valid logger names. Typically, package and class names available in the current project can be auto-completed. Specific frameworks may have extra magic logger names that could be supported as well.

Since a logger name can be any arbitrary name, really, this provider should allow any value but could highlight valid packages and class names that are not available in the project's classpath.

The meta-data snippet below corresponds to the standard `logging.level` property, keys are *logger names* and values correspond to the standard log levels or any custom level:

```
{"hints": [
    {
        "name": "logging.level.keys",
        "values": [
            {
                "value": "root",
                "description": "Root logger used to assign the default logging level."
            }
        ],
        "providers": [
            {
                "name": "logger-name"
            }
        ]
    },
    {
        "name": "logging.level.values",
        "values": [
            {
                "value": "trace"
            },
            {
                "value": "debug"
            },
            {
                "value": "info"
            },
            {
                "value": "warn"
            },
            {
                "value": "error"
            },
            {
                "value": "fatal"
            },
            {
                "value": "off"
            }

        ],
        "providers": [
            {
                "name": "any"
            }
        ]
    }
]}
```

Spring bean reference

The **spring-bean-reference** provider auto-completes the beans that are defined in the configuration of the current project. This provider supports these parameters:

Parameter	Type	Default value	Description
`target`	`String` (`Class`)	*none*	The fully qualified name of the bean class that should be assignable to the candidate. Typically used to filter out non candidate beans.

The meta-data snippet below corresponds to the standard `spring.jmx.server` property that defines the name of the `MBeanServer` bean to use:

```
{"hints": [
    {
        "name": "spring.jmx.server",
        "providers": [
            {
                "name": "spring-bean-reference",
                "parameters": {
                    "target": "javax.management.MBeanServer"
                }
            }
        ]
    }
]}
```

Note

The binder is not aware of the meta-data so if you provide that hint, you will still need to transform the bean name into an actual Bean reference using the `ApplicationContext`.

Spring profile name

The **spring-profile-name** provider auto-completes the Spring profiles that are defined in the configuration of the current project.

The meta-data snippet below corresponds to the standard `spring.profiles.active` property that defines the name of the Spring profile(s) to enable:

```
{"hints": [
    {
        "name": "spring.profiles.active",
        "providers": [
            {
                "name": "spring-profile-name"
            }
        ]
    }
]}
```

B.3 Generating your own 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>
```

With Gradle, you can use the propdeps-plugin and specify:

```
 dependencies {
  optional "org.springframework.boot:spring-boot-configuration-processor"
 }

 compileJava.dependsOn(processResources)
}
```

> **Note**
>
> You need to add `compileJava.dependsOn(processResources)` to your build to ensure that resources are processed before code is compiled. Without this directive any `additional-spring-configuration-metadata.json` files will not be processed.

The processor will pick up 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.

Properties are discovered via the presence of standard getters and setters with special handling for collection types (that will be detected even if only a getter is present). The annotation processor also supports the use of the `@Data`, `@Getter` and `@Setter` lombok annotations.

> **Note**
>
> If you are using AspectJ in your project, you need to make sure that the annotation processor only runs once. There are several ways to do this: with Maven, you can configure the `maven-apt-`

plugin explicitly and add the dependency to the annotation processor only there. You could also let the AspectJ plugin run all the processing and disable annotation processing in the maven-compiler-plugin configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <proc>none</proc>
    </configuration>
</plugin>
```

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 is often the case that properties may exist that are not bound to a `@ConfigurationProperties` bean. You may also need to tune some attributes of an existing key. To support such cases and allow you to provide custom "hints", the annotation processor will automatically merge items from `META-INF/additional-spring-configuration-metadata.json` into the main meta-data file.

If you refer to a property that has been detected automatically, the description, default value and deprecation information are overridden if specified. If the manual property declaration is not identified in the current module, it is added as a brand new property.

The format of the `additional-spring-configuration-metadata.json` file is exactly the same as the regular `spring-configuration-metadata.json`. The additional properties file is optional, if you 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
`ArtemisAutoConfiguration`	javadoc
`BatchAutoConfiguration`	javadoc
`CacheAutoConfiguration`	javadoc
`CassandraAutoConfiguration`	javadoc
`CassandraDataAutoConfiguration`	javadoc
`CassandraRepositoriesAutoConfiguration`	javadoc
`CloudAutoConfiguration`	javadoc
`ConfigurationPropertiesAutoConfiguration`	javadoc
`CouchbaseAutoConfiguration`	javadoc
`CouchbaseDataAutoConfiguration`	javadoc
`CouchbaseRepositoriesAutoConfiguration`	javadoc
`DataSourceAutoConfiguration`	javadoc
`DataSourceTransactionManagerAutoConfiguration`	javadoc
`DeviceDelegatingViewResolverAutoConfiguration`	javadoc
`DeviceResolverAutoConfiguration`	javadoc
`DispatcherServletAutoConfiguration`	javadoc
`ElasticsearchAutoConfiguration`	javadoc
`ElasticsearchDataAutoConfiguration`	javadoc
`ElasticsearchRepositoriesAutoConfiguration`	javadoc
`EmbeddedMongoAutoConfiguration`	javadoc

Configuration Class	Links
`EmbeddedServletContainerAutoConfiguration`	javadoc
`ErrorMvcAutoConfiguration`	javadoc
`FacebookAutoConfiguration`	javadoc
`FallbackWebSecurityAutoConfiguration`	javadoc
`FlywayAutoConfiguration`	javadoc
`FreeMarkerAutoConfiguration`	javadoc
`GroovyTemplateAutoConfiguration`	javadoc
`GsonAutoConfiguration`	javadoc
`H2ConsoleAutoConfiguration`	javadoc
`HazelcastAutoConfiguration`	javadoc
`HazelcastJpaDependencyAutoConfiguration`	javadoc
`HibernateJpaAutoConfiguration`	javadoc
`HornetQAutoConfiguration`	javadoc
`HttpEncodingAutoConfiguration`	javadoc
`HttpMessageConvertersAutoConfiguration`	javadoc
`HypermediaAutoConfiguration`	javadoc
`IntegrationAutoConfiguration`	javadoc
`JacksonAutoConfiguration`	javadoc
`JerseyAutoConfiguration`	javadoc
`JmsAutoConfiguration`	javadoc
`JmxAutoConfiguration`	javadoc
`JndiConnectionFactoryAutoConfiguration`	javadoc
`JndiDataSourceAutoConfiguration`	javadoc
`JooqAutoConfiguration`	javadoc
`JpaRepositoriesAutoConfiguration`	javadoc
`JtaAutoConfiguration`	javadoc
`LinkedInAutoConfiguration`	javadoc
`LiquibaseAutoConfiguration`	javadoc
`MailSenderAutoConfiguration`	javadoc
`MailSenderValidatorAutoConfiguration`	javadoc

Configuration Class	Links
`MessageSourceAutoConfiguration`	javadoc
`MongoAutoConfiguration`	javadoc
`MongoDataAutoConfiguration`	javadoc
`MongoRepositoriesAutoConfiguration`	javadoc
`MultipartAutoConfiguration`	javadoc
`MustacheAutoConfiguration`	javadoc
`Neo4jAutoConfiguration`	javadoc
`Neo4jRepositoriesAutoConfiguration`	javadoc
`OAuth2AutoConfiguration`	javadoc
`PersistenceExceptionTranslationAutoConfiguration`	javadoc
`ProjectInfoAutoConfiguration`	javadoc
`PropertyPlaceholderAutoConfiguration`	javadoc
`RabbitAutoConfiguration`	javadoc
`ReactorAutoConfiguration`	javadoc
`RedisAutoConfiguration`	javadoc
`RedisRepositoriesAutoConfiguration`	javadoc
`RepositoryRestMvcAutoConfiguration`	javadoc
`SecurityAutoConfiguration`	javadoc
`SecurityFilterAutoConfiguration`	javadoc
`SendGridAutoConfiguration`	javadoc
`ServerPropertiesAutoConfiguration`	javadoc
`SessionAutoConfiguration`	javadoc
`SitePreferenceAutoConfiguration`	javadoc
`SocialWebAutoConfiguration`	javadoc
`SolrAutoConfiguration`	javadoc
`SolrRepositoriesAutoConfiguration`	javadoc
`SpringApplicationAdminJmxAutoConfiguration`	javadoc
`SpringDataWebAutoConfiguration`	javadoc
`ThymeleafAutoConfiguration`	javadoc
`TransactionAutoConfiguration`	javadoc

Configuration Class	Links
`TwitterAutoConfiguration`	javadoc
`VelocityAutoConfiguration`	javadoc
`WebMvcAutoConfiguration`	javadoc
`WebSocketAutoConfiguration`	javadoc
`WebSocketMessagingAutoConfiguration`	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
`CacheStatisticsAutoConfiguration`	javadoc
`CrshAutoConfiguration`	javadoc
`EndpointAutoConfiguration`	javadoc
`EndpointMBeanExportAutoConfiguration`	javadoc
`EndpointWebMvcAutoConfiguration`	javadoc
`HealthIndicatorAutoConfiguration`	javadoc
`InfoContributorAutoConfiguration`	javadoc
`JolokiaAutoConfiguration`	javadoc
`ManagementServerPropertiesAutoConfiguration`	javadoc
`ManagementWebSecurityAutoConfiguration`	javadoc
`MetricExportAutoConfiguration`	javadoc
`MetricFilterAutoConfiguration`	javadoc
`MetricRepositoryAutoConfiguration`	javadoc
`MetricsChannelAutoConfiguration`	javadoc
`MetricsDropwizardAutoConfiguration`	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 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>
 +-BOOT-INF
    +-classes
    |  +-mycompany
    |     +-project
    |        +-YourClasses.class
    +-lib
       +-dependency1.jar
       +-dependency2.jar
```

Application classes should be placed in a nested `BOOT-INF/classes` directory. Dependencies should be placed in a nested `BOOT-INF/lib` directory.

The executable war file structure

Spring Boot Loader compatible war files should be structured in the following way:

```
example.war
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-WEB-INF
    +-classes
    |  +-com
    |     +-mycompany
    |        +-project
    |           +-YourClasses.class
    +-lib
    |  +-dependency1.jar
    |  +-dependency2.jar
    +-lib-provided
       +-servlet-api.jar
       +-dependency3.jar
```

Dependencies should be placed in a nested `WEB-INF/lib` directory. Any dependencies that are required when running embedded but are not required when deploying to a traditional web container should be placed in `WEB-INF/lib-provided`.

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 to 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
+-------------------+-------------------------+
| /BOOT-INF/classes | /BOOT-INF/lib/mylib.jar |
|+-----------------+||+-----------+----------+|
||     A.class      |||  B.class  |  C.class ||
|+-----------------+||+-----------+----------+|
+-------------------+-------------------------+
 ^                    ^           ^
 0063                 3452        3980
```

The example above shows how `A.class` can be found in `/BOOT-INF/classes` in `myapp.jar` position `0063`. `B.class` from the nested jar can actually be found in `myapp.jar` position `3452` and `C.class` is at position `3980`.

Armed with this information, we can load specific nested entries by simply seeking to the appropriate part of 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 `getURL()` method will return a `URL` that opens a `java.net.JarURLConnection` compatible connection and 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 `JarLauncher` and `WarLauncher` the nested paths are fixed. `JarLauncher` looks in `BOOT-INF/lib/` and `WarLauncher` looks in `WEB-INF/lib/` and `WEB-INF/lib-provided/` so you just add extra jars in those locations if you want more. The `PropertiesLauncher` looks in `BOOT-INF/lib/` in your application archive 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,${HOME}/app/lib`. Earlier entries take precedence, just like a regular `-classpath` on the `javac` command line.
`loader.home`	Location of additional properties file, e.g. `/opt/app` (defaults to `${user.dir}`)
`loader.args`	Default arguments for the main method (space separated)

Key	Purpose
`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`).
`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`).

> **Tip**
>
> Build plugins automatically move the `Main-Class` attribute to `Start-Class` when the fat jar is built. If you are using that, specify the name of the class to launch using the `Main-Class` attribute and leave out `Start-Class`.

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).
- `loader.path` (if empty) defaults to `lib` (meaning a local directory or a nested one if running from an archive). Because of this `PropertiesLauncher` behaves the same as `JarLauncher` when no additional configuration is provided.
- 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
`antlr`	`antlr`	2.7.7
`ch.qos.logback`	`logback-access`	1.1.7
`ch.qos.logback`	`logback-classic`	1.1.7
`ch.qos.logback`	`logback-core`	1.1.7
`com.atomikos`	`transactions-jdbc`	3.9.3
`com.atomikos`	`transactions-jms`	3.9.3
`com.atomikos`	`transactions-jta`	3.9.3
`com.couchbase.client`	`couchbase-spring-cache`	2.0.0
`com.couchbase.client`	`java-client`	2.2.5
`com.datastax.cassandra`	`cassandra-driver-core`	2.1.9
`com.datastax.cassandra`	`cassandra-driver-dse`	2.1.9
`com.datastax.cassandra`	`cassandra-driver-mapping`	2.1.9
`com.fasterxml.jackson.core`	`jackson-annotations`	2.7.3
`com.fasterxml.jackson.core`	`jackson-core`	2.7.3
`com.fasterxml.jackson.core`	`jackson-databind`	2.7.3
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-csv`	2.7.3
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-xml`	2.7.3
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-yaml`	2.7.3
`com.fasterxml.jackson.datatype`	`jackson-datatype-guava`	2.7.3
`com.fasterxml.jackson.datatype`	`jackson-datatype-hibernate4`	2.7.3
`com.fasterxml.jackson.datatype`	`jackson-datatype-hibernate5`	2.7.3
`com.fasterxml.jackson.datatype`	`jackson-datatype-jdk7`	2.7.3
`com.fasterxml.jackson.datatype`	`jackson-datatype-jdk8`	2.7.3
`com.fasterxml.jackson.datatype`	`jackson-datatype-joda`	2.7.3

Group ID	Artifact ID	Version
com.fasterxml.jackson.datatype	jackson-datatype-json-org	2.7.3
com.fasterxml.jackson.datatype	jackson-datatype-jsr310	2.7.3
com.fasterxml.jackson.jaxrs	jackson-jaxrs-base	2.7.3
com.fasterxml.jackson.jaxrs	jackson-jaxrs-json-provider	2.7.3
com.fasterxml.jackson.module	jackson-module-jaxb-annotations	2.7.3
com.fasterxml.jackson.module	jackson-module-kotlin	2.7.3
com.fasterxml.jackson.module	jackson-module-parameter-names	2.7.3
com.gemstone.gemfire	gemfire	8.2.0
com.github.ben-manes.caffeine	caffeine	2.2.6
com.github.mxab.thymeleaf.extras	thymeleaf-extras-data-attribute	1.3
com.google.appengine	appengine-api-1.0-sdk	1.9.34
com.google.code.gson	gson	2.6.2
com.googlecode.json-simple	json-simple	1.1.1
com.h2database	h2	1.4.191
com.hazelcast	hazelcast	3.6.1
com.hazelcast	hazelcast-hibernate4	3.6.1
com.hazelcast	hazelcast-spring	3.6.1
com.jayway.jsonpath	json-path	2.2.0
com.jayway.jsonpath	json-path-assert	2.2.0
com.samskivert	jmustache	1.12
com.sendgrid	sendgrid-java	2.2.2
com.sun.mail	javax.mail	1.5.5
com.timgroup	java-statsd-client	3.1.0
com.zaxxer	HikariCP	2.4.5
com.zaxxer	HikariCP-java6	2.3.13
commons-beanutils	commons-beanutils	1.9.2

Group ID	Artifact ID	Version
commons-codec	commons-codec	1.10
commons-collections	commons-collections	3.2.2
commons-dbcp	commons-dbcp	1.4
commons-digester	commons-digester	2.1
commons-pool	commons-pool	1.6
de.flapdoodle.embed	de.flapdoodle.embed.mongo	1.50.2
dom4j	dom4j	1.6.1
io.dropwizard.metrics	metrics-core	3.1.2
io.dropwizard.metrics	metrics-ganglia	3.1.2
io.dropwizard.metrics	metrics-graphite	3.1.2
io.dropwizard.metrics	metrics-servlets	3.1.2
io.projectreactor	reactor-bus	2.0.7.RELEASE
io.projectreactor	reactor-core	2.0.7.RELEASE
io.projectreactor	reactor-groovy	2.0.7.RELEASE
io.projectreactor	reactor-groovy-extensions	2.0.7.RELEASE
io.projectreactor	reactor-logback	2.0.7.RELEASE
io.projectreactor	reactor-net	2.0.7.RELEASE
io.projectreactor	reactor-stream	2.0.7.RELEASE
io.projectreactor.spring	reactor-spring-context	2.0.7.RELEASE
io.projectreactor.spring	reactor-spring-core	2.0.7.RELEASE
io.projectreactor.spring	reactor-spring-messaging	2.0.7.RELEASE
io.projectreactor.spring	reactor-spring-webmvc	2.0.7.RELEASE
io.undertow	undertow-core	1.3.20.Final
io.undertow	undertow-servlet	1.3.20.Final
io.undertow	undertow-websockets-jsr	1.3.20.Final
javax.cache	cache-api	1.0.0
javax.jms	jms-api	1.1-rev-1
javax.mail	javax.mail-api	1.5.5
javax.servlet	javax.servlet-api	3.1.0

Group ID	Artifact ID	Version
javax.servlet	jstl	1.2
javax.transaction	javax.transaction-api	1.2
jaxen	jaxen	1.1.6
joda-time	joda-time	2.9.3
junit	junit	4.12
mysql	mysql-connector-java	5.1.38
net.java.dev.jna	jna	4.2.2
net.sf.ehcache	ehcache	2.10.1
net.sourceforge.htmlunit	htmlunit	2.20
net.sourceforge.jtds	jtds	1.3.1
net.sourceforge.nekohtml	nekohtml	1.9.22
nz.net.ultraq.thymeleaf	thymeleaf-layout-dialect	1.3.3
org.apache.activemq	activemq-amqp	5.13.2
org.apache.activemq	activemq-blueprint	5.13.2
org.apache.activemq	activemq-broker	5.13.2
org.apache.activemq	activemq-camel	5.13.2
org.apache.activemq	activemq-client	5.13.2
org.apache.activemq	activemq-console	5.13.2
org.apache.activemq	activemq-http	5.13.2
org.apache.activemq	activemq-jaas	5.13.2
org.apache.activemq	activemq-jdbc-store	5.13.2
org.apache.activemq	activemq-jms-pool	5.13.2
org.apache.activemq	activemq-kahadb-store	5.13.2
org.apache.activemq	activemq-karaf	5.13.2
org.apache.activemq	activemq-leveldb-store	5.13.2
org.apache.activemq	activemq-log4j-appender	5.13.2
org.apache.activemq	activemq-mqtt	5.13.2
org.apache.activemq	activemq-openwire-generator	5.13.2

Group ID	Artifact ID	Version
org.apache.activemq	activemq-openwire-legacy	5.13.2
org.apache.activemq	activemq-osgi	5.13.2
org.apache.activemq	activemq-partition	5.13.2
org.apache.activemq	activemq-pool	5.13.2
org.apache.activemq	activemq-ra	5.13.2
org.apache.activemq	activemq-run	5.13.2
org.apache.activemq	activemq-runtime-config	5.13.2
org.apache.activemq	activemq-shiro	5.13.2
org.apache.activemq	activemq-spring	5.13.2
org.apache.activemq	activemq-stomp	5.13.2
org.apache.activemq	activemq-web	5.13.2
org.apache.activemq	artemis-jms-client	1.2.0
org.apache.activemq	artemis-jms-server	1.2.0
org.apache.commons	commons-dbcp2	2.1.1
org.apache.commons	commons-pool2	2.4.2
org.apache.derby	derby	10.12.1.1
org.apache.httpcomponents	httpasyncclient	4.1.1
org.apache.httpcomponents	httpclient	4.5.2
org.apache.httpcomponents	httpcore	4.4.4
org.apache.httpcomponents	httpmime	4.5.2
org.apache.logging.log4j	log4j-api	2.5
org.apache.logging.log4j	log4j-core	2.5
org.apache.logging.log4j	log4j-slf4j-impl	2.5
org.apache.solr	solr-solrj	5.5.0
org.apache.tomcat	tomcat-jdbc	8.0.33
org.apache.tomcat	tomcat-jsp-api	8.0.33
org.apache.tomcat.embed	tomcat-embed-core	8.0.33
org.apache.tomcat.embed	tomcat-embed-el	8.0.33
org.apache.tomcat.embed	tomcat-embed-jasper	8.0.33

Group ID	Artifact ID	Version
org.apache.tomcat.embed	tomcat-embed-logging-juli	8.0.33
org.apache.tomcat.embed	tomcat-embed-websocket	8.0.33
org.apache.velocity	velocity	1.7
org.apache.velocity	velocity-tools	2.0
org.aspectj	aspectjrt	1.8.9
org.aspectj	aspectjtools	1.8.9
org.aspectj	aspectjweaver	1.8.9
org.assertj	assertj-core	2.4.1
org.codehaus.btm	btm	2.1.4
org.codehaus.groovy	groovy	2.4.6
org.codehaus.groovy	groovy-all	2.4.6
org.codehaus.groovy	groovy-ant	2.4.6
org.codehaus.groovy	groovy-bsf	2.4.6
org.codehaus.groovy	groovy-console	2.4.6
org.codehaus.groovy	groovy-docgenerator	2.4.6
org.codehaus.groovy	groovy-groovydoc	2.4.6
org.codehaus.groovy	groovy-groovysh	2.4.6
org.codehaus.groovy	groovy-jmx	2.4.6
org.codehaus.groovy	groovy-json	2.4.6
org.codehaus.groovy	groovy-jsr223	2.4.6
org.codehaus.groovy	groovy-nio	2.4.6
org.codehaus.groovy	groovy-servlet	2.4.6
org.codehaus.groovy	groovy-sql	2.4.6
org.codehaus.groovy	groovy-swing	2.4.6
org.codehaus.groovy	groovy-templates	2.4.6
org.codehaus.groovy	groovy-test	2.4.6
org.codehaus.groovy	groovy-testng	2.4.6
org.codehaus.groovy	groovy-xml	2.4.6
org.codehaus.janino	janino	2.7.8
org.crashub	crash.cli	1.3.2

Group ID	Artifact ID	Version
`org.crashub`	`crash.connectors.ssh`	1.3.2
`org.crashub`	`crash.connectors.telnet`	1.3.2
`org.crashub`	`crash.embed.spring`	1.3.2
`org.crashub`	`crash.plugins.cron`	1.3.2
`org.crashub`	`crash.plugins.mail`	1.3.2
`org.crashub`	`crash.shell`	1.3.2
`org.eclipse.jetty`	`jetty-annotations`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-client`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-continuation`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-deploy`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-http`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-io`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-jmx`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-jsp`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-plus`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-security`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-server`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-servlet`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-servlets`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-util`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-webapp`	9.2.15.v20160210
`org.eclipse.jetty`	`jetty-xml`	9.2.15.v20160210
`org.eclipse.jetty.orbit`	`javax.servlet.jsp`	2.2.0.v201112011158
`org.eclipse.jetty.websocket`	`javax-websocket-server-impl`	9.2.15.v20160210
`org.eclipse.jetty.websocket`	`websocket-client`	9.2.15.v20160210
`org.eclipse.jetty.websocket`	`websocket-server`	9.2.15.v20160210
`org.elasticsearch`	`elasticsearch`	2.2.2
`org.firebirdsql.jdbc`	`jaybird-jdk16`	2.2.10
`org.firebirdsql.jdbc`	`jaybird-jdk17`	2.2.10
`org.firebirdsql.jdbc`	`jaybird-jdk18`	2.2.10

Group ID	Artifact ID	Version
org.flywaydb	flyway-core	3.2.1
org.freemarker	freemarker	2.3.23
org.glassfish	javax.el	3.0.0
org.glassfish.jersey.containers	jersey-container-servlet	2.22.2
org.glassfish.jersey.containers	jersey-container-servlet-core	2.22.2
org.glassfish.jersey.core	jersey-server	2.22.2
org.glassfish.jersey.ext	jersey-bean-validation	2.22.2
org.glassfish.jersey.ext	jersey-spring3	2.22.2
org.glassfish.jersey.media	jersey-media-json-jackson	2.22.2
org.hamcrest	hamcrest-core	1.3
org.hamcrest	hamcrest-library	1.3
org.hibernate	hibernate-core	5.1.0.Final
org.hibernate	hibernate-ehcache	5.1.0.Final
org.hibernate	hibernate-entitymanager	5.1.0.Final
org.hibernate	hibernate-envers	5.1.0.Final
org.hibernate	hibernate-java8	5.1.0.Final
org.hibernate	hibernate-jpamodelgen	5.1.0.Final
org.hibernate	hibernate-validator	5.2.4.Final
org.hibernate	hibernate-validator-annotation-processor	5.2.4.Final
org.hornetq	hornetq-jms-client	2.4.7.Final
org.hornetq	hornetq-jms-server	2.4.7.Final
org.hsqldb	hsqldb	2.3.3
org.infinispan	infinispan-jcache	8.1.3.Final
org.infinispan	infinispan-spring4-common	8.1.3.Final
org.infinispan	infinispan-spring4-embedded	8.1.3.Final
org.javassist	javassist	3.18.1-GA

Group ID	Artifact ID	Version
`org.jboss`	`jboss-transaction-spi`	7.3.0.Final
`org.jboss.logging`	`jboss-logging`	3.3.0.Final
`org.jboss.narayana.jta`	`jdbc`	5.3.2.Final
`org.jboss.narayana.jta`	`jms`	5.3.2.Final
`org.jboss.narayana.jta`	`jta`	5.3.2.Final
`org.jboss.narayana.jts`	`narayana-jts-integration`	5.3.2.Final
`org.jdom`	`jdom2`	2.0.6
`org.jolokia`	`jolokia-core`	1.3.3
`org.jooq`	`jooq`	3.7.3
`org.jooq`	`jooq-codegen`	3.7.3
`org.jooq`	`jooq-meta`	3.7.3
`org.json`	`json`	20140107
`org.liquibase`	`liquibase-core`	3.4.2
`org.mariadb.jdbc`	`mariadb-java-client`	1.4.0
`org.mockito`	`mockito-core`	1.10.19
`org.mongodb`	`mongo-java-driver`	2.14.2
`org.neo4j`	`neo4j-ogm-api`	2.0.0
`org.neo4j`	`neo4j-ogm-compiler`	2.0.0
`org.neo4j`	`neo4j-ogm-core`	2.0.0
`org.neo4j`	`neo4j-ogm-http-driver`	2.0.0
`org.postgresql`	`postgresql`	9.4.1208.jre7
`org.projectlombok`	`lombok`	1.16.8
`org.seleniumhq.selenium`	`selenium-api`	2.52.0
`org.seleniumhq.selenium`	`selenium-htmlunit-driver`	2.52.0
`org.seleniumhq.selenium`	`selenium-remote-driver`	2.52.0
`org.seleniumhq.selenium`	`selenium-support`	2.52.0
`org.skyscreamer`	`jsonassert`	1.3.0
`org.slf4j`	`jcl-over-slf4j`	1.7.21
`org.slf4j`	`jul-to-slf4j`	1.7.21

Group ID	Artifact ID	Version
`org.slf4j`	`log4j-over-slf4j`	1.7.21
`org.slf4j`	`slf4j-api`	1.7.21
`org.slf4j`	`slf4j-jdk14`	1.7.21
`org.slf4j`	`slf4j-log4j12`	1.7.21
`org.slf4j`	`slf4j-simple`	1.7.21
`org.spockframework`	`spock-core`	1.0-groovy-2.4
`org.spockframework`	`spock-spring`	1.0-groovy-2.4
`org.springframework`	`spring-aop`	4.3.0.RC1
`org.springframework`	`spring-aspects`	4.3.0.RC1
`org.springframework`	`spring-beans`	4.3.0.RC1
`org.springframework`	`spring-context`	4.3.0.RC1
`org.springframework`	`spring-context-support`	4.3.0.RC1
`org.springframework`	`spring-core`	4.3.0.RC1
`org.springframework`	`spring-expression`	4.3.0.RC1
`org.springframework`	`spring-instrument`	4.3.0.RC1
`org.springframework`	`spring-instrument-tomcat`	4.3.0.RC1
`org.springframework`	`spring-jdbc`	4.3.0.RC1
`org.springframework`	`spring-jms`	4.3.0.RC1
`org.springframework`	`springloaded`	1.2.5.RELEASE
`org.springframework`	`spring-messaging`	4.3.0.RC1
`org.springframework`	`spring-orm`	4.3.0.RC1
`org.springframework`	`spring-oxm`	4.3.0.RC1
`org.springframework`	`spring-test`	4.3.0.RC1
`org.springframework`	`spring-tx`	4.3.0.RC1
`org.springframework`	`spring-web`	4.3.0.RC1
`org.springframework`	`spring-webmvc`	4.3.0.RC1
`org.springframework`	`spring-webmvc-portlet`	4.3.0.RC1
`org.springframework`	`spring-websocket`	4.3.0.RC1
`org.springframework.amqp`	`spring-amqp`	1.6.0.M2
`org.springframework.amqp`	`spring-rabbit`	1.6.0.M2

Group ID	Artifact ID	Version
org.springframework.batch	spring-batch-core	3.0.6.RELEASE
org.springframework.batch	spring-batch-infrastructure	3.0.6.RELEASE
org.springframework.batch	spring-batch-integration	3.0.6.RELEASE
org.springframework.batch	spring-batch-test	3.0.6.RELEASE
org.springframework.boot	spring-boot	1.4.0.M2
org.springframework.boot	spring-boot	1.4.0.M2
org.springframework.boot	spring-boot-actuator	1.4.0.M2
org.springframework.boot	spring-boot-actuator-docs	1.4.0.M2
org.springframework.boot	spring-boot-autoconfigure	1.4.0.M2
org.springframework.boot	spring-boot-configuration-metadata	1.4.0.M2
org.springframework.boot	spring-boot-configuration-processor	1.4.0.M2
org.springframework.boot	spring-boot-devtools	1.4.0.M2
org.springframework.boot	spring-boot-loader	1.4.0.M2
org.springframework.boot	spring-boot-loader-tools	1.4.0.M2
org.springframework.boot	spring-boot-starter	1.4.0.M2
org.springframework.boot	spring-boot-starter-actuator	1.4.0.M2
org.springframework.boot	spring-boot-starter-amqp	1.4.0.M2
org.springframework.boot	spring-boot-starter-aop	1.4.0.M2
org.springframework.boot	spring-boot-starter-artemis	1.4.0.M2
org.springframework.boot	spring-boot-starter-batch	1.4.0.M2
org.springframework.boot	spring-boot-starter-cache	1.4.0.M2
org.springframework.boot	spring-boot-starter-cloud-connectors	1.4.0.M2

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-data-cassandra	1.4.0.M2
org.springframework.boot	spring-boot-starter-data-couchbase	1.4.0.M2
org.springframework.boot	spring-boot-starter-data-elasticsearch	1.4.0.M2
org.springframework.boot	spring-boot-starter-data-gemfire	1.4.0.M2
org.springframework.boot	spring-boot-starter-data-jpa	1.4.0.M2
org.springframework.boot	spring-boot-starter-data-mongodb	1.4.0.M2
org.springframework.boot	spring-boot-starter-data-neo4j	1.4.0.M2
org.springframework.boot	spring-boot-starter-data-redis	1.4.0.M2
org.springframework.boot	spring-boot-starter-data-rest	1.4.0.M2
org.springframework.boot	spring-boot-starter-data-solr	1.4.0.M2
org.springframework.boot	spring-boot-starter-freemarker	1.4.0.M2
org.springframework.boot	spring-boot-starter-groovy-templates	1.4.0.M2
org.springframework.boot	spring-boot-starter-hateoas	1.4.0.M2
org.springframework.boot	spring-boot-starter-hornetq	1.4.0.M2
org.springframework.boot	spring-boot-starter-integration	1.4.0.M2
org.springframework.boot	spring-boot-starter-jdbc	1.4.0.M2
org.springframework.boot	spring-boot-starter-jersey	1.4.0.M2
org.springframework.boot	spring-boot-starter-jetty	1.4.0.M2

Group ID	Artifact ID	Version
`org.springframework.boot`	`spring-boot-starter-jooq`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-jta-atomikos`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-jta-bitronix`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-jta-narayana`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-log4j2`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-logging`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-mail`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-mobile`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-mustache`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-remote-shell`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-security`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-social-facebook`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-social-linkedin`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-social-twitter`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-test`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-thymeleaf`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-tomcat`	1.4.0.M2
`org.springframework.boot`	`spring-boot-starter-undertow`	1.4.0.M2

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-validation	1.4.0.M2
org.springframework.boot	spring-boot-starter-velocity	1.4.0.M2
org.springframework.boot	spring-boot-starter-web	1.4.0.M2
org.springframework.boot	spring-boot-starter-websocket	1.4.0.M2
org.springframework.boot	spring-boot-starter-ws	1.4.0.M2
org.springframework.boot	spring-boot-test	1.4.0.M2
org.springframework.boot	spring-boot-test	1.4.0.M2
org.springframework.boot	spring-boot-test-autoconfigure	1.4.0.M2
org.springframework.cloud	spring-cloud-cloudfoundry-connector	1.2.2.RELEASE
org.springframework.cloud	spring-cloud-core	1.2.2.RELEASE
org.springframework.cloud	spring-cloud-heroku-connector	1.2.2.RELEASE
org.springframework.cloud	spring-cloud-localconfig-connector	1.2.2.RELEASE
org.springframework.cloud	spring-cloud-spring-service-connector	1.2.2.RELEASE
org.springframework.data	spring-cql	1.4.1.RELEASE
org.springframework.data	spring-data-cassandra	1.4.1.RELEASE
org.springframework.data	spring-data-commons	1.12.1.RELEASE
org.springframework.data	spring-data-couchbase	2.1.1.RELEASE
org.springframework.data	spring-data-elasticsearch	2.0.1.RELEASE
org.springframework.data	spring-data-envers	1.0.1.RELEASE
org.springframework.data	spring-data-gemfire	1.8.1.RELEASE
org.springframework.data	spring-data-jpa	1.10.1.RELEASE
org.springframework.data	spring-data-keyvalue	1.1.1.RELEASE
org.springframework.data	spring-data-mongodb	1.9.1.RELEASE
org.springframework.data	spring-data-mongodb-cross-store	1.9.1.RELEASE

Group ID	Artifact ID	Version
`org.springframework.data`	`spring-data-mongodb-log4j`	1.9.1.RELEASE
`org.springframework.data`	`spring-data-neo4j`	4.1.1.RELEASE
`org.springframework.data`	`spring-data-redis`	1.7.1.RELEASE
`org.springframework.data`	`spring-data-rest-core`	2.5.1.RELEASE
`org.springframework.data`	`spring-data-rest-hal-browser`	2.5.1.RELEASE
`org.springframework.data`	`spring-data-rest-webmvc`	2.5.1.RELEASE
`org.springframework.data`	`spring-data-solr`	2.0.1.RELEASE
`org.springframework.hateoas`	`spring-hateoas`	0.19.0.RELEASE
`org.springframework.integration`	`spring-integration-amqp`	4.3.0.M1
`org.springframework.integration`	`spring-integration-core`	4.3.0.M1
`org.springframework.integration`	`spring-integration-event`	4.3.0.M1
`org.springframework.integration`	`spring-integration-feed`	4.3.0.M1
`org.springframework.integration`	`spring-integration-file`	4.3.0.M1
`org.springframework.integration`	`spring-integration-ftp`	4.3.0.M1
`org.springframework.integration`	`spring-integration-gemfire`	4.3.0.M1
`org.springframework.integration`	`spring-integration-groovy`	4.3.0.M1
`org.springframework.integration`	`spring-integration-http`	4.3.0.M1
`org.springframework.integration`	`spring-integration-ip`	4.3.0.M1
`org.springframework.integration`	`spring-integration-jdbc`	4.3.0.M1
`org.springframework.integration`	`spring-integration-jms`	4.3.0.M1
`org.springframework.integration`	`spring-integration-jmx`	4.3.0.M1
`org.springframework.integration`	`spring-integration-jpa`	4.3.0.M1
`org.springframework.integration`	`spring-integration-mail`	4.3.0.M1
`org.springframework.integration`	`spring-integration-mongodb`	4.3.0.M1
`org.springframework.integration`	`spring-integration-mqtt`	4.3.0.M1
`org.springframework.integration`	`spring-integration-redis`	4.3.0.M1

Group ID	Artifact ID	Version
org.springframework.integration	spring-integration-rmi	4.3.0.M1
org.springframework.integration	spring-integration-scripting	4.3.0.M1
org.springframework.integration	spring-integration-security	4.3.0.M1
org.springframework.integration	spring-integration-sftp	4.3.0.M1
org.springframework.integration	spring-integration-stomp	4.3.0.M1
org.springframework.integration	spring-integration-stream	4.3.0.M1
org.springframework.integration	spring-integration-syslog	4.3.0.M1
org.springframework.integration	spring-integration-test	4.3.0.M1
org.springframework.integration	spring-integration-twitter	4.3.0.M1
org.springframework.integration	spring-integration-websocket	4.3.0.M1
org.springframework.integration	spring-integration-ws	4.3.0.M1
org.springframework.integration	spring-integration-xml	4.3.0.M1
org.springframework.integration	spring-integration-xmpp	4.3.0.M1
org.springframework.integration	spring-integration-zookeeper	4.3.0.M1
org.springframework.mobile	spring-mobile-device	1.1.5.RELEASE
org.springframework.plugin	spring-plugin-core	1.2.0.RELEASE
org.springframework.restdocs	spring-restdocs-core	1.1.0.M1
org.springframework.restdocs	spring-restdocs-mockmvc	1.1.0.M1
org.springframework.restdocs	spring-restdocs-restassured	1.1.0.M1
org.springframework.retry	spring-retry	1.1.2.RELEASE
org.springframework.security	spring-security-acl	4.0.4.RELEASE
org.springframework.security	spring-security-aspects	4.0.4.RELEASE
org.springframework.security	spring-security-cas	4.0.4.RELEASE
org.springframework.security	spring-security-config	4.0.4.RELEASE

Group ID	Artifact ID	Version
org.springframework.security	spring-security-core	4.0.4.RELEASE
org.springframework.security	spring-security-crypto	4.0.4.RELEASE
org.springframework.security	spring-security-data	4.0.4.RELEASE
org.springframework.security	spring-security-jwt	1.0.4.RELEASE
org.springframework.security	spring-security-ldap	4.0.4.RELEASE
org.springframework.security	spring-security-messaging	4.0.4.RELEASE
org.springframework.security	spring-security-openid	4.0.4.RELEASE
org.springframework.security	spring-security-remoting	4.0.4.RELEASE
org.springframework.security	spring-security-taglibs	4.0.4.RELEASE
org.springframework.security	spring-security-test	4.0.4.RELEASE
org.springframework.security	spring-security-web	4.0.4.RELEASE
org.springframework.security.oauth	spring-security-oauth	2.0.9.RELEASE
org.springframework.security.oauth	spring-security-oauth2	2.0.9.RELEASE
org.springframework.session	spring-session	1.2.0.RC2
org.springframework.session	spring-session-data-gemfire	1.2.0.RC2
org.springframework.session	spring-session-data-mongo	1.2.0.RC2
org.springframework.session	spring-session-data-redis	1.2.0.RC2
org.springframework.session	spring-session-jdbc	1.2.0.RC2
org.springframework.social	spring-social-config	1.1.4.RELEASE
org.springframework.social	spring-social-core	1.1.4.RELEASE
org.springframework.social	spring-social-facebook	2.0.3.RELEASE
org.springframework.social	spring-social-facebook-web	2.0.3.RELEASE
org.springframework.social	spring-social-linkedin	1.0.2.RELEASE
org.springframework.social	spring-social-security	1.1.4.RELEASE
org.springframework.social	spring-social-twitter	1.1.2.RELEASE
org.springframework.social	spring-social-web	1.1.4.RELEASE

Group ID	Artifact ID	Version
`org.springframework.ws`	`spring-ws-core`	2.3.0.RELEASE
`org.springframework.ws`	`spring-ws-security`	2.3.0.RELEASE
`org.springframework.ws`	`spring-ws-support`	2.3.0.RELEASE
`org.springframework.ws`	`spring-ws-test`	2.3.0.RELEASE
`org.thymeleaf`	`thymeleaf`	2.1.4.RELEASE
`org.thymeleaf`	`thymeleaf-spring4`	2.1.4.RELEASE
`org.thymeleaf.extras`	`thymeleaf-extras-conditionalcomments`	2.1.1.RELEASE
`org.thymeleaf.extras`	`thymeleaf-extras-java8time`	2.1.0.RELEASE
`org.thymeleaf.extras`	`thymeleaf-extras-springsecurity4`	2.1.2.RELEASE
`org.webjars`	`hal-browser`	9f96c74
`org.webjars`	`webjars-locator`	0.30
`org.xerial`	`sqlite-jdbc`	3.8.11.2
`org.yaml`	`snakeyaml`	1.17
`redis.clients`	`jedis`	2.8.1
`wsdl4j`	`wsdl4j`	1.6.3