Spring Boot Reference Guide

1.0.1.RELEASE

Phillip Webb , Dave Syer , Josh Long , Stéphane Nicoll , Rob Winch

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 http://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 is builds on many other Spring projects, check the spring.io web-site for a wealth of reference documentation. If you are just starting out with Spring, try one of the guides.
- Ask a questions - we monitor stackoverflow.com for questions tagged with `spring-boot`.
- Report bugs with Spring Boot at https://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 your just getting started with Spring Boot, or *Spring* in general, this is the place to start!

- **From scratch:** Overview | 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
- **Testing:** Overview | Boot Applications | Utils
- **Extending:** Auto-configuration | @Conditions

6. Moving to production

When your 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

7. Advanced topics

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

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

Part II. Getting started

If your 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 can 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. 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.

9.1 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 3.0 or above. If you don’t already have Maven installed you can follow the instructions at http://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>1.0.1.RELEASE</version>
    </parent>

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

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

</project>
```

Gradle installation

Spring Boot is compatible with Gradle 1.6 or above. If you don't already have Gradle installed you can follow the instructions at http://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 http://www.gradle.org/docs/current/userguide/gradle_wrapper.html for details.

Here is a typical `build.gradle` file:

```
buildscript {
    repositories {
        mavenCentral()
        maven { url "http://repo.spring.io/snapshot" }
        maven { url "http://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.1.RELEASE")
    }
}

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

jar {
    baseName = 'myproject'
    version =  '0.0.1-SNAPSHOT'
}

repositories {
    mavenCentral()
```

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

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

9.2 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-1.0.1.RELEASE-bin.zip
- spring-boot-cli-1.0.1.RELEASE-bin.tar.gz

Cutting edge snapshot distributions are also available.

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

Installation with GVM

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

```
$ gvm install springboot
$ spring --version
Spring Boot v1.0.1.RELEASE
```

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

```
$ gvm install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-cli-1.0.1.RELEASE-
bin/spring-1.0.1.RELEASE/
$ gvm use springboot dev
$ spring --version
Spring CLI v1.0.1.RELEASE
```

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

You can see it by doing this:

```
$ gvm ls springboot

================================================================================
```

```
Available Springboot Versions
================================================================================
> + dev
* 1.0.1.RELEASE

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

OSX Homebrew installation

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

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

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

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

Command-line completion

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

```
$ . ~/.gvm/springboot/current/bash_completion.d/spring
$ spring <HIT TAB HERE>
  grab  help  jar  run  test  version
```

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

Quick start Spring CLI example

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

```
@Controller
class ThisWillActuallyRun {

    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "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 http://localhost:8080 in your favorite web browser and you should see the following output:

```
Hello World!
```

10. 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.

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 3.1.1 (0728685237757ffbf44136acec0402957f723d9a; 2013-09-17 08:22:22-0700)
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".

10.1 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 you 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>1.0.1.RELEASE</version>
    </parent>

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

</project>
```

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

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

10.2 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
[INFO] +- junit:junit:jar:4.11:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] +- org.mockito:mockito-core:jar:1.9.5:test
[INFO] |  \- org.objenesis:objenesis:jar:1.0:test
[INFO] \- org.hamcrest:hamcrest-library:jar:1.3:test
```

The `mvn dependency:tree` command prints tree representation of your project dependencies. You can see that `spring-boot-starter-parent` has already provided some useful test dependencies. 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.

10.3 Writing the code

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

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

@Controller
@EnableAutoConfiguration
public class Example {

    @RequestMapping("/")
    @ResponseBody
    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 though the important parts.

The @Controller, @RequestMapping and @ResponseBody annotations

The first annotation on our `Example` class is `@Controller`. 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 additional `@ResponseBody` annotation tells Spring to render the resulting string directly back to the caller.

> **Tip**
>
> The `@Controller`, `@RequestMapping` and `@ResponseBody` 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 though to expose any command-line arguments.

10.4 Running the example

At this point out 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 ::  (v1.0.1.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.514)
```

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

```
Hello World!
```

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

10.5 Creating an executable jar

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

Executable jars and Java

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

To solve this problem, many developers use "shaded" jars. A shaded jar simply packages all classes, from all jars, into a single "uber jar". The problem with shaded jars is that it becomes hard to see which libraries you are actually using in your application. It can also be problematic if the the same filename is used (but with different content) in multiple jars.

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

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

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

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:1.0.1.RELEASE:repackage (default) @ myproject ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

If you look in the `target` directory you should see `myproject-0.0.1-SNAPSHOT.jar`. The file should be around 10 Mb in size. If you want to peek inside, you can use `jar tvf`:

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

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

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

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

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.0.1.RELEASE)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 3.236 seconds (JVM running for 3.764)
```

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

11. 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 your a task oriented type of developer you might want to jump over to http://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 a Spring Boot specific *How-to* reference documentation.

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

Part III. Using Spring Boot

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

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

12. 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.

12.1 Maven

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

- Java 1.6 as the default compiler level.
- UTF-8 source encoding.
- A Dependency Management section, allowing you to omit `<version>` tags for common dependencies.
- Generally useful test dependencies (JUnit, Hamcrest, Mockito).
- Sensible resource filtering.
- Sensible plugin configuration (exec plugin, surefire, Git commit ID, shade).

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>1.0.1.RELEASE</version>
</parent>
```

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

Using your own parent POM

If you don’t want to use the Spring Boot starter parent, you can use your own and still keep the benefit of the dependency management (but not the plugin management) using a `scope=import` dependency:

```
<dependencyManagement>
     <dependencies>
        <dependency>
            <!-- Import dependency management from Spring Boot -->
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>1.0.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
```

```
</dependencyManagement>
```

Changing the Java version

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

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

Using the Spring Boot Maven plugin

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

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

> **Note**
>
> 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.

12.2 Gradle

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

```
apply plugin: 'java'

repositories { mavenCentral() }
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.0.1.RELEASE")
}
```

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

```
buildscript {
    repositories { mavenCentral() }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.1.RELEASE")
    }
}

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

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

12.3 Ant

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

See the *Section 64.1, "Build an executable archive with Ant"* "How-to" for more complete instructions.

12.4 Starter POMs

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

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

What's in a name

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

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

Table 12.1. Spring Boot application starters

Name	Description
`spring-boot-starter`	The core Spring Boot starter, including auto-configuration support, logging and YAML.
`spring-boot-starter-amqp`	Support for the "Advanced Message Queuing Protocol" via `spring-rabbit`.
`spring-boot-starter-aop`	Full AOP programming support including `spring-aop` and AspectJ.
`spring-boot-starter-batch`	Support for "Spring Batch" including HSQLDB database.
`spring-boot-starter-data-jpa`	Full support for the "Java Persistence API" including `spring-data-jpa`, `spring-orm` and Hibernate.
`spring-boot-starter-data-mongodb`	Support for the MongoDB NoSQL Database, including `spring-data-mongodb`.

Name	Description
`spring-boot-starter-data-rest`	Support for exposing Spring Data repositories over REST via `spring-data-rest-webmvc`.
`spring-boot-starter-integration`	Support for common `spring-integration` modules.
`spring-boot-starter-jdbc`	JDBC Database support.
`spring-boot-starter-mobile`	Support for `spring-mobile`
`spring-boot-starter-redis`	Support for the REDIS key-value data store, including `spring-redis`.
`spring-boot-starter-security`	Support for `spring-security`.
`spring-boot-starter-test`	Support for common test dependencies, including JUnit, Hamcrest and Mockito along with the `spring-test` module.
`spring-boot-starter-thymeleaf`	Support for the Thymeleaf templating engine, including integration with Spring.
`spring-boot-starter-web`	Support for full-stack web development, including Tomcat and `spring-webmvc`.
`spring-boot-starter-websocket`	Support for websocket development with Tomcat.

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

Table 12.2. Spring Boot production ready starters

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

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

Table 12.3. Spring Boot technical starters

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

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

13. Structuring your code

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

13.1 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` or `@EntityScan` 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`).

13.2 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.

Here is a typical layout:

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

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

```
package com.example.myproject;

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

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

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

}
```

14. 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-base configuration if possible. Searching for `enable*` annotations can be a good starting point.

14.1 Importing additional configuration classes

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

14.2 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.

15. 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` annotation 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.

15.1 Gradually replacing auto-configuration

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

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

15.2 Disabling specific auto-configuration

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

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

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

16. 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.

17. 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.

17.1 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 meta-data 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 `Relauch` button rather than `Run` to ensure that any existing instance is closed.

17.2 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
```

17.3 Using the Maven plugin

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

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

17.4 Using the Gradle plugin

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

```
$ gradle bootRun
```

17.5 Hot swapping

Since Spring Boot applications are just plain Java application, JVM hot-swapping should work out of the box. JVM hot swapping is somewhat limited with the bytecode that it can replace, for a more complete solution the Spring Loaded project, or JRebel can be used.

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

18. 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, “Production-ready features”* for details.

19. 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.

20. 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 ::   v1.0.1.RELEASE

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

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

20.1 Customizing SpringApplication

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

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

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

It is also possible to configure the `SpringApplication` using an `application.properties` file. See *Chapter 21, Externalized Configuration* for details.

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

20.2 Fluent builder API

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

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

For example:

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

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

20.3 Application events and listeners

In addition to the usual Spring Framework events, such as `ContextRefreshedEvent`, a `SpringApplication` sends some additional application events. Some events are actually triggered before the `ApplicationContext` is created.

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

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

1. An `ApplicationStartedEvent` is sent at the start of a run, but before any processing except the registration of listeners and initializers.
2. An `ApplicationEnvironmentPreparedEvent` is sent when the `Environment` to be used in the context is known, but before the context is created.
3. An `ApplicationPreparedEvent` is sent just before the refresh is started, but after bean definitions have been loaded.
4. An `ApplicationFailedEvent` is sent if there is an exception on startup.

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

20.4 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.

20.5 Using the CommandLineRunner

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

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

@Component
public class MyBean implements CommandLineRunner {

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

}
```

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

20.6 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.

21. Externalized Configuration

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

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

1. Command line arguments.
2. Java System properties (`System.getProperties()`).
3. OS environment variables.
4. `@PropertySource` annotations on your `@Configuration` classes.
5. Application properties outside of your packaged jar (`application.properties` including YAML and profile variants).
6. Application properties packaged inside your jar (`application.properties` including YAML and profile variants).
7. Default properties (specified using `SpringApplication.setDefaultProperties`).

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

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

@Component
public class MyBean {

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

    // ...

}
```

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

21.1 Accessing command line properties

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

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

21.2 Application property files

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

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

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

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

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

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

or

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

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

21.3 Profile specific properties

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

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

21.4 Placeholders in properties

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

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

> **Tip**
>
> You can also use this technique to create “short” variants of existing Spring Boot properties. See the *Section 54.3, “Use “short” command line arguments”* how-to for details.

21.5 Using YAML instead of Properties

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

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

Loading YAML

Spring Boot 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:

```
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 comma-separated values (useful for simple String values) and also as property keys with `[index]` dereferencers, for example this YAML:

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

Would be transformed into these properties:

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

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 document in a single file by 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
```

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.

21.6 Typesafe Configuration Properties

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

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

    private String username;

    private InetAddress remoteAddress;

    // ... getters and setters

}
```

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

```
# application.yml

connection:
    username: admin
    remoteAddress: 192.168.1.1

# additional configuration as required
```

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

```
@Service
public class MyService {
```

```
    @Autowired
    private ConnectionSettings connection;

     //...

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

}
```

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

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

Relaxed binding

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

Spring will attempt to coerce the external application properties to the right type when it binds to the `@ConfigurationProperties` beans. If you need custom type conversion you can provide a `ConversionService` bean (with bean id `conversionService`) or custom property editors (via a `CustomEditorConfigurer` bean).

@ConfigurationProperties Validation

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

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

    @NotNull
    private InetAddress remoteAddress;

    // ... getters and setters

}
```

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

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

22. 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 ProductionConfiguraiton {

    // ...

}
```

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`.

22.1 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
```

22.2 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.

22.3 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 21.3, "Profile specific properties"* for details.

23. 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, Log4J and Logback. In each case there is console output and file output (rotating, 10 Mb file size).

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.

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

23.2 Console output

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

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

If your terminal supports ANSI, color output will be used to aid readability.

23.3 File output

By default, log files are written to `spring.log` in your `temp` directory and rotate at 10 Mb. You can easily customize the output folder by setting the `logging.path` property (for example in your `application.properties`). It is also possible to change the filename using a `logging.file` property.

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

23.4 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`.

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

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

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

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

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

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

24. Developing web applications

Spring Boot is well suited for web application development. You can easily create a self-contained HTTP server using embedded Tomcat or Jetty. 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.

24.1 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 http://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).
- Static `index.html` support.
- Custom `Favicon` support.

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

HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box, for example Objects can be automatically converted to JSON (using the Jackson library) or XML (using JAXB).

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);
    }

}
```

Static Content

By default Spring Boot will serve static content from a folder called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath or from the root of the `ServeltContext`. It uses the `ResourceHttpRequestHandler` from Spring MVC so you can modify that behavior by adding your own `WebMvcConfigurerAdapter` and overriding the `addResourceHandlers` method.

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

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

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

Template engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies including: velocity, freemarker, and JSPs. Many other templating engines also ship their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the Thymeleaf templating engine. Thymeleaf is an XML/XHTML/HTML5 template engine that can work both in web and non-web environments. If allows you to create natural templates, that can be correctly displayed by browsers and therefore work also as static prototypes. Thymeleaf templates will be picked up automatically from `src/main/resources/templates`.

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

24.2 Embedded servlet container support

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

Servlets and Filters

When using an embedded servlet container you can register Servlets and Filters directly as Spring beans. This can be particularly convenient if you want to refer to a value from your `application.properties` during configuration.

By default, if the context contains only a single Servlet it will be mapped to `/`. In the case of multiple Servlets beans the bean name will be used as a path prefix. Filters will map to `/*`.

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

The EmbeddedWebApplicationContext

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

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

Customizing embedded servlet containers

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

Common server settings include:

- `server.port` — The listen port for incoming HTTP requests.

- `server.address` — The interface address to bind to.
- `server.sessionTimeout` — A session timeout.

See the `ServerProperties` class for a complete list.

Programmatic customization

If you need to configure your embdedded servlet container programmatically you can register a Spring bean that implements the `EmbeddedServletContainerCustomizer` interface. `EmbeddedServletContainerCustomizer` provides access to the `ConfigurableEmbeddedServletContainerFactory` 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 ConfigurableEmbeddedServletContainerFactory directly

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

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

Setters are provided for many configuration options. Several protected method “hooks” are also provided should you need to do something more exotic. See the source code documentation for details.

JSP limitations

When running a Spring Boot application that uses an embedded servlet container (and is packaged as an executable archive), there are some limitations in the JSP support.

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

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

25. 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 (username “user” and password random, printed at INFO level when the application starts up). You can change the password by providing a `security.user.password`. This and other useful properties are externalized via `SecurityProperties` (properties prefix "security").

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

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

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

All of the above can be switched on and off or modified using external properties (`security.*`).

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

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

The Actuator security features can be modified using external properties (`management.security.*`).

26. 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.

26.1 Configure a DataSource

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

Embedded Database Support

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

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

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

For example, typical POM dependencies would be:

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

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

Connection to a production database

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

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

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

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

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

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

See `AbstractDataSourceConfiguration` for more of the supported options.

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

26.2 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;
    }

    // ...

}
```

26.3 JPA and "Spring Data"

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

- Hibernate — One of the most popular JPA implementations.
- Spring Data JPA — Makes it easy to easily implement JPA based repositories.

- Spring ORMs — Core ORM support from the Spring Framework.

> **Tip**
>
> We won't go into too many details of JPA or Spring Data here. You can follow the "Accessing Data with JPA" guide from http://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`) 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 58.3, "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`) 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 database 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). Alternatively, `spring.jpa.generate-ddl=false` switches off all DDL generation.

27. Working with NoSQL technologies

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

27.1 MongoDB

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

Connecting to a MongoDB database

You can inject an auto-configured `com.mongodb.Mongo` instance as you would any other Spring Bean. By default the instance will attempt to connect to a MongoDB server using the URL `mongodb://localhost/test`:

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

import com.mongodb.Mongo;

@Component
public class MyBean {

    private final Mongo mongo;

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

    // ...

}
```

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

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

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

You can also declare your own `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.

28. Testing

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

28.1 Test scope dependencies

If you extend your Maven project from the `spring-boot-starter-parent` POM, or 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.
- Hamcrest — A library of matcher objects (also known as constraints or predicates) allowing `assertThat` style JUnit assertions.
- Mockito — A Java mocking framework.

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

28.2 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 use the `spring-test` module before you should start by reading the relevant section of the Spring Framework reference documentation.

28.3 Testing Spring Boot applications

A Spring Boot application is just a Spring `ApplicationContext` so nothing very special has to be done to test it beyond what you would normally do with a vanilla Spring context. One thing to watch out for though is that the external properties, logging and other features of Spring Boot are only installed in the context by default if you use `SpringApplication` to create it.

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

For example:

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

    @Autowired
    CityRepository repository;

    // ...

}
```

Tip

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

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

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

    @Autowired
    CityRepository repository;

    RestTemplate restTemplate = new TestRestTemplate();

    // ... interact with the running server

}
```

28.4 Test utilities

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

ConfigFileApplicationContextInitializer

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

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

EnvironmentTestUtils

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

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

OutputCapture

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

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.OutputCapture;

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

public class MyTest {

 @Rule
 public OutputCapture capture = new OutputCapture();

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

}
```

TestRestTemplate

`TestRestTemplate` is a convenience subclass of Spring's `RestTemplate` that is useful in integration tests. You can get a vanilla template or one that sends Basic HTTP authentication (with a username and password). And in either case the template will behave in a friendly way for testing, 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"));
 }

}
```

29. Developing auto-configuration and using conditions

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.

29.1 Understanding auto-configured beans

Under the hood, auto-configuration is implemented with standard `@Configuration` classes. Additional `@Conditional` annotations are used to constrain when the auto-configuration should apply. Usually auto-configuration classes use `@ConditionalOnClass` and `@ConditionalOnMissingBean` annotations. This ensures that auto-configuration 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).

29.2 Locating auto-configuration candidates

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

```
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`.

29.3 Condition annotations

You almost always want to include one or more `@Condition` annotations on your auto-configuration class. The `@ConditionalOnMissingBean` is one common example that is used to allow developers to “override” auto-configuration if they are not happy with your defaults.

Spring Boot includes a number of `@Conditional` annotations that you can reuse in your own code by annotating `@Configuration` classes or individual `@Bean` methods.

Class conditions

The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations allows configuration to be skipped based on the presence or absence of specific classes. Due to the fact that annotation meta-data is parsed using ASM you can actually use the `value` attribute to refer to the real class, even though that class might not actually appear on the running application classpath. You can also use the `name` attribute if you prefer to specify the class name using a `String` value.

Bean conditions

The `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations allow configurations to be skipped based on the presence or absence of specific beans. You can use the `value` attribute to

specify beans by type, or `name` to specify beans by name. The `search` attribute allows you to limit the `ApplicationContext` hierarchy that should be considered when searching for beans.

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

Resource conditions

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

Web Application Conditions

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

SpEL expression conditions

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

30. 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. 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.

31. 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")
}
```

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

The following endpoints are available:

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

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

32.1 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.

32.2 Custom health information

The default information exposed by the `health` endpoint is a simple “OK” message. It is often useful to perform some additional health checks, for example you might check that a database connection works, or that a remote REST endpoint is functioning.

To provide custom health information you can register a Spring bean that implements the `HealthIndicator` interface.

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

@Component
public class MyHealth implements HealthIndicator<String> {

    @Override
    public String health() {
        // perform some specific health check
        return ...
    }

}
```

Spring Boot also provides a `SimpleHealthIndicator` implementation that attempts a simple database test.

32.3 Custom application info information

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

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

If you are using Maven, you can automatically expand info properties from the Maven project using resource filtering. In your `pom.xml` you have (inside the `<build/>` element):

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

You can then refer to your Maven “project properties” via placeholders, e.g.

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

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

Git commit information

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

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

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

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

33. Monitoring and management over HTTP

If you are developing a Spring MVC application, Spring Boot Actuator will auto-configure all non-sensitive 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`.

33.1 Exposing sensitive endpoints

If you use “Spring Security” sensitive endpoints will be exposed over HTTP, but also 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 password for application endpoints”.

You can use Spring properties to change the username and passsword 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
```

33.2 Customizing the management server context path

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

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

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

33.3 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.)

33.4 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
```

33.5 Disabling HTTP endpoints

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

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

34. 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.

34.1 Customizing MBean names

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

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

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

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

34.2 Disabling JMX endpoints

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

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

34.3 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
```

35. 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 a dependency to `spring-boot-starter-remote-shell`:

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

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

35.1 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, 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 ::  (v1.0.1.RELEASE) on myhost
```

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

Remote shell credentials

You can use the `shell.auth.simple.username` and `shell.auth.simple.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.

35.2 Extending the remote shell

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

Remote shell commands

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

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

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

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

```
package commands

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

class hello {

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

}
```

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

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

Remote shell plugins

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

For more information please refer to the CRaSH reference documentation.

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

```
{
    "counter.status.200.root": 20,
    "counter.status.200.metrics": 3,
    "counter.status.401.root": 4,
    "gauge.response.root": 2,
    "gauge.response.metrics": 3,
    "mem": 466944,
    "mem.free": 410117,
    "processors": 8
}
```

Here we can see basic `memory` and `processor` 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 `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.

36.1 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.

36.2 Metric repositories

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

36.3 Coda Hale Metrics

User of the Coda Hale "Metrics" library will automatically find that Spring Boot metrics are published to `com.codahale.metrics.MetricRegistry`. A default `com.codahale.metrics.MetricRegistry` Spring bean will be created when you declare a dependency to the `com.codahale.metrics:metrics-core` library; you can also register you own `@Bean` instance if you need customizations.

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

36.4 Message channel integration

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

37. 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.

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`).

38. 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,
    ...
}]
```

38.1 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.

39. Error Handling

Spring Boot Actuator provides an `/error` mapping by default that handles all errors in a sensible way. If you want more specific error pages for some conditions, the embedded servlet containers support a uniform Java DSL for customizing the error handling.

40. 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 "cloud deployment options" or jump ahead for some in depth information about Spring Boot's *build tool plugins*.

Part VI. Deploying to the cloud

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

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

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

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

41. 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 install`) and installed the `cf` command line tool, simply answer the `cf push` command prompts as follows, substituting the path to your compiled `.jar` for mine. Be sure to have logged in with your `cf` command line client before attempting to use it.

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

If there is a Cloud Foundry `manifest.yml` file present in the same directory, it will be consulted. If not, the client will prompt you with questions it has about how it should deploy and manage your application, starting with its name:

```
Name> acloudyspringtime

Instances> 1

1: 128M
2: 256M
3: 512M
4: 1G
Memory Limit> 256M

Creating acloudyspringtime... OK

1: acloudyspringtime
2: none
Subdomain> acloudyspringtime

1: cfapps.io
2: none
Domain> cfapps.io

Creating route acloudyspringtime.cfapps.io... OK
Binding acloudyspringtime.cfapps.io to acloudyspringtime... OK

Create services for application?> n

Bind other services to application?> n

Save configuration?> y

Saving to manifest.yml... OK
```

> **Note**
>
> Here we are substituting `acloudyspringtime` for whatever value you give `cf` when it asks for 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)
Push successful! App 'acloudyspringtime' available at acloudyspringtime.cfapps.io
```

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                            status    usage      url
...
acloudyspringtime               running   1 x 256M   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/`.

41.1 Binding to services

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

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

```
@Component
class MyBean implements EnvironmentAware {

    private String instanceId;

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

    // ...

}
```

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

> **Tip**
>
> The Spring Cloud project is a better fit for tasks such as configuring a DataSource; and you can also use Spring Cloud with Heroku too!

42. 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. This is a breeze with Spring Boot. 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 or Jetty instance which then uses it when it starts up. The `$PORT` environment variable is assigned to us by the Heroku PaaS.

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

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

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

```
$ git push heroku master

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

-----> Java app detected
-----> Installing OpenJDK 1.7... done
-----> Installing Maven 3.0.3... done
-----> Installing settings.xml... done
-----> executing /app/tmp/cache/.maven/bin/mvn -B
       -Duser.home=/tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229
       -Dmaven.repo.local=/app/tmp/cache/.m2/repository
       -s /app/tmp/cache/.m2/settings.xml -DskipTests=true clean install

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

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

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

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

That should be it! Your application should be up and running on Heroku.

43. CloudBees

CloudBees provides cloud-based “continuous integration” and “continuous delivery” services as well as Java PaaS hosting. Sean Gilligan has contributed an excellent Spring Boot sample application to the CloudBees community GitHub repository. The project includes an extensive README that covers the steps that you need to follow when deploying to CloudBees.

44. What to read next

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

45. Installing the CLI

The Spring Boot CLI can be installed manually; using GVM (the Groovy Environment Manually) or using Homebrew if you are an OSX user. See *Section 9.2, "Installing the Spring Boot CLI"* in the "Getting started" section for comprehensive installation instructions.

46. 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 v1.0.1.RELEASE
```

46.1 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:

```
@Controller
class WebApplication {

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

}
```

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 `@Controller` annotations, “Tomcat” and “Spring MVC” will be grabbed.

The following items are used as “grab hints”:

Items	Grabs
`JdbcTemplate`, `NamedParameterJdbcTemplate`, `DataSource`	JDBC Application.
`@EnableJmsMessaging`	JMS Application.
`@Test`	JUnit.
`@EnableRabbitMessaging`	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.

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`, `@Controller`, `@RequestMapping` and `@ResponseBody` 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 equilvement 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`.

46.2 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 `test.groovy` file that we used above:

```
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.

46.3 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
```

46.4 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.

See the output of `spring help jar` for more information.

46.5 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 (v1.0.1.RELEASE)
Hit TAB to complete. Type 'help' and hit RETURN for help, and 'exit' to quit.
```

From inside the embedded shell you can run other commands directly:

```
$ version
Spring CLI v1.0.1.RELEASE
```

The embedded shell supports ANSI color output as well as `tab` completion. If you need to run a native command you can use the $ prefix. Hitting `ctrl-c` will exit the embedded shell.

47. 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.

48. 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 12, *Build systems*” from the Part III, “Using Spring Boot” section first.

49. 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 (or better).

49.1 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>1.0.1.RELEASE</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

This configuration will repackage a jar or war that is built during the `package` phase of the Maven lifecycle. The following example shows both the repackaged jar, as well as the original jar, in the `target` directory:

```
$ mvn package
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

If you don’t include the `<execution/>` configuration as above, you can run the plugin on its own (but only if the package goal is used as well). For example:

```
$ mvn package spring-boot:repackage
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

If you are using a milestone or snapshot release you will also need to add appropriate `pluginRepository` elements:

```
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <url>http://repo.spring.io/snapshot</url>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <url>http://repo.spring.io/milestone</url>
    </pluginRepository>
</pluginRepositories>
```

49.2 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>
```

49.3 Repackage configuration

The following configuration options are available for the `spring-boot:repackage` goal:

Required parameters

Name	Description
`outputDirectory`	Directory containing the generated archive (defaults to `${project.build.directory}`).
`finalName`	Name of the generated archive (defaults to `${project.build.finalName}`).

Optional parameters

Name	Description
`classifier`	Classifier to add to the generated artifact. If given, the artifact will be attached. If this is not given, it will merely be written to the output directory according to the `finalName`. Attaching the artifact allows to deploy it alongside to the original one, see the maven documentation for more details
`mainClass`	The name of the main class. If not specified will search for a single compiled class that contains a `main` method.
`layout`	The type of archive (which corresponds to how the dependencies are laid out inside it). Defaults to a guess based on the archive type.

The plugin rewrites your manifest, and in particular it manages the `Main-Class` and `Start-Class` entries, so if the defaults don't work you have to configure those there (not in the jar plugin). The `Main-Class` in the manifest is actually controlled by the `layout` property of the boot plugin, e.g.

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>1.0.1.RELEASE</version>
    <configuration>
        <mainClass>${start-class}</mainClass>
        <layout>ZIP</layout>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>repackage</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

The layout property defaults to a guess based on the archive type (jar or war). For the `PropertiesLauncher` the layout is "ZIP" (even though the output might be a jar file).

> **Tip**
>
> The executable jar format is described in the appendix.

49.4 Running applications

The Spring Boot Maven Plugin includes a `run` goal which can be used to launch your application from the command line. Type the following from the root of your Maven project:

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

By default, any `src/main/resources` folder will be added to the application classpath when you run via the maven plugin. This allows hot refreshing of resources which can be very useful when developing web applications. For example, you can work on HTML, CSS or JavaScipt files and see your changes immediately without recompiling your application. It is also a helpful way of allowing your front end developers to work without needing to download and install a Java IDE.

49.5 Run configuration

The following configuration options are available for the `spring-boot:run` goal:

49.6 Required parameters

Name	Description
`classesDirectrory`	Directory containing the classes and resource files that should be packaged into the archive (defaults to `${project.build.outputDirectory}`).

49.7 Optional parameters

Name	Description
`arguments` or `-Drun.arguments`	Arguments that should be passed to the application.
`addResources` or `-Drun.addResources`	Add Maven resources to the classpath directly, this allows live in-place editing or resources. Since resources will be added directly, and via the target/classes folder they will appear twice if `ClassLoader.getResources()` is called. In practice, however, most applications call `ClassLoader.getResource()` which will always return the first resource (defaults to `true`).
`mainClass`	The name of the main class. If not specified the first compiled class found that contains a *main* method will be used.
`folders`	Folders that should be added to the classpath (defaults to `${project.build.outputDirectory}`).

50. 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 omit version information from your `build.gradle` file for “blessed” dependencies.

50.1 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:1.0.1.RELEASE")
    }
}
apply plugin: 'spring-boot'
```

If you are using a milestone or snapshot release you will also need to add appropriate `repositories` reference:

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

50.2 Declaring dependencies without versions

The `spring-boot` plugin will register a custom Gradle `ResolutionStrategy` with your build that allows you to omit version numbers when declaring dependencies to “blessed” artifacts. All artifacts with a `org.springframework.boot` group ID, and any of the artifacts declared in the `managementDependencies` section of the `spring-dependencies` POM can have their version number resolved automatically.

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")
}
```

50.3 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.

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 {
    mavenCentral()
    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")
    ...
}
```

50.4 Running a project in-place

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

```
$ gradle bootRun
```

Running this way makes your static classpath resources (i.e. in `src/main/resources` by default) reloadable in the live application, which can be helpful at development time.

50.5 Repackage configuration

The gradle plugin automatically extends your build script DSL with a `springBoot` element for configuration. Simply set the appropriate properties as you would with any other Gradle extension (see below for a list of configuration options):

```
springBoot {
    backupSource = false
}
```

50.6 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. If not specified the value from the manifest will be used, or if no manifest entry is the archive will be searched for a suitable class.
`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).

50.7 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 disables 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.

51. Supporting other build systems

If you want to use a build tool other than Maven or Gradle, you will likely need to develop your own plugin. Executable jars need to follow a specific format and certain entries need to be written in an uncompressed form (see the *executable jar format* section in the appendix for details).

The Spring Boot Maven and Gradle plugins both make use of `spring-boot-loader-tools` to actually generate jars. You are also free to use this library directly yourself if you need to.

51.1 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.

51.2 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`.

51.3 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.

51.4 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(nestedFile, LibraryScope.COMPILE);
            }
        });
```

52. What to read next

If your interested to looking at how the build tool plugins were developed 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.

53. Spring Boot application

53.1 Troubleshoot auto-configuration

The Spring Boot auto-configuration tries it's best to "do the right thing", but sometimes things fail and it can be hard to tell why.

There is a really useful `AutoConfigurationReport` available in any Spring Boot `ApplicationContext`. You will see it if you enable `DEBUG` logging output. If you use the `spring-boot-actuator` there is also an `autoconfig` endpoint that renders the report in JSON. Use that to debug the application and see what features have been added (and which not) by Spring Boot at runtime.

Many more questions can be answered by looking at the source code and the javadoc. Some rules of thumb:

- Look for classes called `*AutoConfiguration` and read their sources, in particular the `@Conditional*` annotations to find out what features they enable and when. Add `--debug` to the command line or a System property `-Ddebug` to get a log on the console of all the autoconfiguration decisions that were made in your app. In a running Actuator app look at the `autoconfig` endpoint ('/autoconfig' or the JMX equivalent) for the same information.
- Look for classes that are `@ConfigurationProperties` (e.g. `ServerProperties` and read from there the available external configuration options. The `@ConfigurationProperties` has a `name` attribute which acts as a prefix to external properties, thus `ServerProperties` has `prefix="server"` and its configuration properties are `server.port`, `server.address` etc. In a running Actuator app look at the `configprops` endpoint.
- Look for use of `RelaxedEnvironment` to pull configuration values explicitly out of the `Environment`. It often is used with a prefix.
- Look for `@Value` annotations that bind directly to the `Environment`. This is less flexible than the `RelaxedEnvironment` approach, but does allow some relaxed binding, specifically for OS environment variables (so `CAPITALS_AND_UNDERSCORES` are synonyms for `period.separated`).
- Look for `@ConditionalOnExpression` annotations that switch features on and off in response to SpEL expressions, normally evaluated with place-holders resolved from the `Environment`.

53.2 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 20.3, "Application events and listeners"* in the "Spring Boot features" section for a complete list.

53.3 Build an ApplicationContext hierarchy (adding a parent or root context)

You can use the `ApplicationBuilder` class to create parent/child `ApplicationContext` hierarchies. See *Section 20.2, "Fluent builder API"* in the "Spring Boot features" section for more information.

53.4 Create a non-web application

Not all Spring applications have to be web applications (or web services). If you want to execute some code in a `main` method, but also bootstrap a Spring application to set up the infrastructure to use, then it's easy with the `SpringApplication` features of Spring Boot. A `SpringApplication` changes its `ApplicationContext` class depending on whether it thinks it needs a web application or not. The first thing you can do to help it is to just leave the servlet API dependencies off the classpath. If you can't do that (e.g. you are running 2 applications from the same code base) then you can explicitly call `SpringApplication.setWebEnvironment(false)`, or set the `applicationContextClass` property (through the Java API or with external properties). Application code that you want to run as your business logic can be implemented as a `CommandLineRunner` and dropped into the context as a `@Bean` definition.

54. Properties & configuration

54.1 Externalize the configuration of SpringApplication

A `SpringApplication` has bean properties (mainly setters) so you can use its Java API as you create the application to modify its behavior. Or you can externalize the configuration using properties in `spring.main.*`. E.g. in `application.properties` you might have.

```
spring.main.web_environment=false
spring.main.show_banner=false
```

and then the Spring Boot banner will not be printed on startup, and the application will not be a web application.

> **Note**
>
> The example above also demonstrates how flexible binding allows the use of underscores (_) as well as dashes (-) in property names.

54.2 Change the location of external properties of an application

By default properties from different sources are added to the Spring `Environment` in a defined order (see *Chapter 21, Externalized Configuration* in the “Spring Boot features” section for the exact order).

A nice way to augment and modify this is to add `@PropertySource` annotations to your application sources. Classes passed to the `SpringApplication` static convenience methods, and those added using `setSources()` are inspected to see if they have `@PropertySources`, and if they do, those properties are added to the `Environment` early enough to be used in all phases of the `ApplicationContext` lifecycle. Properties added in this way have precedence over any added using the default locations, but have lower priority than system properties, environment variables or the command line.

You can also provide System properties (or environment variables) to change the behavior:

- `spring.config.name` (`SPRING_CONFIG_NAME`), defaults to `application` as the root of the file name.
- `spring.config.location` (`SPRING_CONFIG_LOCATION`) is the file to load (e.g. a classpath resource or a URL). A separate `Environment` property source is set up for this document and it can be overridden by system properties, environment variables or the command line.

No matter what you set in the environment, Spring Boot will always load `application.properties` as described above. If YAML is used then files with the “.yml” extension are also added to the list by default.

See `ConfigFileApplicationListener` for more detail.

54.3 Use “short” command line arguments

Some people like to use (for example) `--port=9000` instead of `--server.port=9000` to set configuration properties on the command line. You can easily enable this by using placeholders in `application.properties`, e.g.

```
server.port=${port:8080}
```

> **Tip**
>
> If you have enabled maven filtering for the `application.properties` you may want to avoid using `${*}` for the tokens to filter as it conflicts with those placeholders. You can either use `@*@` (i.e. `@maven.token@` instead of `${maven.token}`) or you can configure the `maven-resources-plugin` 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.

54.4 Use YAML for external properties

YAML is a superset of JSON and as such is a very convenient syntax for storing external properties in a hierarchical format. E.g.

```
spring:
    application:
        name: cruncher
    datasource:
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost/test
server:
    port: 9000
```

Create a file called `application.yml` and stick it in the root of your classpath, and also add `snakeyaml` to your dependencies (Maven coordinates `org.yaml:snakeyaml`, already included if you use the `spring-boot-starter`). A YAML file is parsed to a Java `Map<String,Object>` (like a JSON object), and Spring Boot flattens the map so that it is 1-level deep and has period-separated keys, a lot like people are used to with `Properties` files in Java.

The example YAML above corresponds to an `application.properties` file

```
spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

See *Section 21.5, "Using YAML instead of Properties"* in the "Spring Boot features" section for more information about YAML.

54.5 Set the active Spring profiles

The Spring `Environment` has an API for this, but normally you would set a System profile (`spring.profiles.active`) or an OS environment variable (`SPRING_PROFILES_ACTIVE`). E.g. launch your application with a `-D` argument (remember to put it before the main class or jar archive):

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

In Spring Boot you can also set the active profile in `application.properties`, e.g.

```
spring.profiles.active=production
```

A value set this way is replaced by the System property or environment variable setting, but not by the `SpringApplicationBuilder.profiles()` method. Thus the latter Java API can be used to augment the profiles without changing the defaults.

See *Chapter 22, Profiles* in the “Spring Boot features” section for more information.

54.6 Change configuration depending on the environment

A YAML file is actually a sequence of documents separated by `---` lines, and each document is parsed separately to a flattened map.

If a YAML document contains a `spring.profiles` key, then the profiles value (comma-separated list of profiles) is fed into the Spring `Environment.acceptsProfiles()` and if any of those profiles is active that document is included in the final merge (otherwise not).

Example:

```
server:
    port: 9000
---

spring:
    profiles: development
server:
    port: 9001

---

spring:
    profiles: production
server:
    port: 0
```

In this example the default port is 9000, but if the Spring profile “development” is active then the port is 9001, and if “production” is active then it is 0.

The YAML documents are merged in the order they are encountered (so later values override earlier ones).

To do the same thing with properties files you can use `application-${profile}.properties` to specify profile-specific values.

54.7 Discover built-in options for external properties

Spring Boot binds external properties from `application.properties` (or `.yml`) (and other places) into an application at runtime. There is not (and technically cannot be) an exhaustive list of all supported properties in a single location because contributions can come from additional jar files on your classpath.

A running application with the Actuator features has a `configprops` endpoint that shows all the bound and bindable properties available through `@ConfigurationProperties`.

The appendix includes an `application.properties` example with a list of the most common properties supported by Spring Boot. The definitive list comes from searching the source code for `@ConfigurationProperties` and `@Value` annotations, as well as the occasional use of `RelaxedEnvironment`.

55. Embedded servlet containers

55.1 Add a Servlet, Filter or ServletContextListener to an application

`Servlet`, `Filter`, `ServletContextListener` and the other listeners supported by the Servlet spec can be added to your application as `@Bean` definitions. Be very careful that they don't cause eager initialization of too many other beans because they have to be installed in the container very early in the application lifecycle (e.g. it's not a good idea to have them depend on your `DataSource` or JPA configuration). You can work around restrictions like that by initializing them lazily when first used instead of on initialization.

In the case of `Filters` and `Servlets` you can also add mappings and init parameters by adding a `FilterRegistrationBean` or `ServletRegistrationBean` instead of or as well as the underlying component.

55.2 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.

55.3 Use a random unassigned HTTP port

To scan for a free port (using OS natives to prevent clashes) use `server.port=0`.

55.4 Discover the HTTP port at runtime

You can access the port the server is running on from log output or from the `EmbeddedWebApplicationContext` via its `EmbeddedServletContainer`. The best way to get that and be sure that it has initialized is to add a `@Bean` of type `ApplicationListener<EmbeddedServletContainerInitializedEvent>` and pull the container out of the event when it is published.

A really useful thing to do in is to autowire the `EmbeddedWebApplicationContext` into a test case and use it to discover the port that the app is running on. In that way you can use a test profile that chooses a random port (`server.port=0`) and make your test suite independent of its environment. Example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SampleDataJpaApplication.class)
@WebApplication
@IntegrationTest
@ActiveProfiles("test")
public class CityRepositoryIntegrationTests {

    @Autowired
```

```
    EmbeddedWebApplicationContext server;

    int port;

    @Before
    public void init() {
        port = server.getEmbeddedServletContainer().getPort();
    }

    // ...

}
```

55.5 Configure Tomcat

Generally you can follow the advice from *Section 54.7, "Discover built-in options for external properties"* about `@ConfigurationProperties` (`ServerProperties` is the main one here), but also look at `EmbeddedServletContainerCustomizer` and various Tomcat specific `*Customizers` that you can add in one of those. The Tomcat APIs are quite rich so once you have access to the `TomcatEmbeddedServletContainerFactory` you can modify it in a number of ways. Or the nuclear option is to add your own `TomcatEmbeddedServletContainerFactory`.

55.6 Terminate SSL in Tomcat

Use an `EmbeddedServletContainerCustomizer` and in that add a `TomcatConnectorCustomizer` that sets up the connector to be secure:

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

// ...

private static class MyCustomizer implements EmbeddedServletContainerCustomizer {

    @Override
    public void customize(ConfigurableEmbeddedServletContainer factory) {
        if(factory instanceof TomcatEmbeddedServletContainerFactory) {
            customizeTomcat((TomcatEmbeddedServletContainerFactory) factory));
        }
    }

    public void customizeTomcat(TomcatEmbeddedServletContainerFactory factory) {
        factory.addConnectorCustomizers(new TomcatConnectorCustomizer() {
            @Override
            public void customize(Connector connector) {
                connector.setPort(serverPort);
                connector.setSecure(true);
                connector.setScheme("https");
                connector.setAttribute("keyAlias", "tomcat");
                connector.setAttribute("keystorePass", "password");
                try {
                    connector.setAttribute("keystoreFile",
                        ResourceUtils.getFile("src/ssl/tomcat.keystore").getAbsolutePath());
                } catch (FileNotFoundException e) {
                    throw new IllegalStateException("Cannot load keystore", e);
                }
                connector.setAttribute("clientAuth", "false");
                connector.setAttribute("sslProtocol", "TLS");
                connector.setAttribute("SSLEnabled", true);
            }
        });
    }
```

```
}
```

55.7 Enable Multiple Connectors Tomcat

Add a `org.apache.catalina.connector.Connector` to the `TomcatEmbeddedServletContainerFactory` which can allow multiple connectors eg a 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);
    }
}
```

55.8 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:1.0.0.RC3")
    compile("org.springframework.boot:spring-boot-starter-jetty:1.0.0.RC3")
    // ...
}
```

55.9 Configure Jetty

Generally you can follow the advice from *Section 54.7, "Discover built-in options for external properties"* about `@ConfigurationProperties` (`ServerProperties` is the main one here), but also look at `EmbeddedServletContainerCustomizer`. The Jetty APIs are quite rich so once you have access to the `JettyEmbeddedServletContainerFactory` you can modify it in a number of ways. Or the nuclear option is to add your own `JettyEmbeddedServletContainerFactory`.

55.10 Use Tomcat 8

Tomcat 8 works with Spring Boot, but the default is to use Tomcat 7 (so we can support Java 1.6 out of the box). You should only need to change the classpath to use Tomcat 8 for it to work. For example, using the starter poms in Maven:

```
<properties>
    <tomcat.version>8.0.3</tomcat.version>
</properties>
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    ...
</dependencies>
```

change the classpath to use Tomcat 8 for it to work.

55.11 Use Jetty 9

Jetty 9 works with Spring Boot, but the default is to use Jetty 8 (so we can support Java 1.6 out of the box). You should only need to change the classpath to use Jetty 9 for it to work.

If you are using the starter poms and parent you can just add the Jetty starter and change the version properties, e.g. for a simple webapp or service:

```
<properties>
    <java.version>1.7</java.version>
    <jetty.version>9.1.0.v20131115</jetty.version>
    <servlet-api.version>3.1.0</servlet-api.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifactId>
            </exclusion>
        </exclusions>
```

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

56. Spring MVC

56.1 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 `http://localhost:8080/thing` will serve a JSON representation of it by default. Sometimes in a browser you might see XML responses (but by default only if `MyThing` was a JAXB object) because browsers tend to send accept headers that prefer XML.

56.2 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 a default converter with a vanilla `ObjectMapper`. Spring Boot has some features to make it easier to customize this behavior.

The smallest change that might work is to just add beans of type `com.fasterxml.jackson.databind.Module` to your context. They will be registered with the default `ObjectMapper` and then injected into the default message converter. To replace the default `ObjectMapper` completely, define a `@Bean` of that type and mark it as `@Primary`.

In addition, if your context contains any beans of type `ObjectMapper` then all of the `Module` beans will be registered with all of the mappers. So there is 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 56.3, "Customize the @ResponseBody rendering"* section and the `WebMvcAutoConfiguration` source code for more details.

56.3 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.

56.4 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).

56.5 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.

56.6 Customize ViewResolvers

A `ViewResolver` is a core component of Spring MVC, translating view names in `@Controller` to actual `View` implementations. Note that `ViewResolvers` are mainly used in UI applications, rather than REST-style services (a `View` is not used to render a `@ResponseBody`). There are many implementations of `ViewResolver` to choose from, and Spring on its own is not opinionated about which ones you should use. Spring Boot, on the other hand, installs one or two for you depending on what it finds on the classpath and in the application context. The `DispatcherServlet` uses all the resolvers it finds in the application context, trying each one in turn until it gets a result, so if you are adding your own you have to be aware of the order and in which position your resolver is added.

`WebMvcAutoConfiguration` adds the following `ViewResolvers` to your context:

- An `InternalResourceViewResolver` with bean id “defaultViewResolver”. This one locates physical resources that can be rendered using the `DefaultServlet` (e.g. static resources and JSP pages if you are using those). It applies a prefix and a suffix to the view name and then looks for a physical resource with that path in the servlet context (defaults are both empty, but accessible for external configuration via `spring.view.prefix` and `spring.view.suffix`). It can be overridden by providing a bean of the same type.
- A `BeanNameViewResolver` with id “beanNameViewResolver”. This is a useful member of the view resolver chain and will pick up any beans with the same name as the `View` being resolved. It can be overridden by providing a bean of the same type, but it’s unlikely you will need to do that.
- 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.

Checkout `WebMvcAutoConfiguration` and `ThymeleafAutoConfiguration`

57. Logging

57.1 Configure Logback for logging

Spring Boot has no mandatory logging dependence, except for the `commons-logging` API, of which there are many implementations to choose from. To use Logback you need to include it, and some bindings for `commons-logging` on the classpath. The simplest way to do that is through the starter poms which all depend on `spring-boot-starter-logging`. For a web application you only need `spring-boot-starter-web` since it depends transitively on the logging starter. For example, using Maven:

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

Spring Boot has a `LoggingSystem` abstraction that attempts to configure logging based on the content of the classpath. If Logback is available it is the first choice. So if you put a `logback.xml` in the root of your classpath it will be picked up from there. Spring Boot provides a default base configuration that you can include if you just want to set levels, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>
    <logger name="org.springframework.web" level="DEBUG"/>
</configuration>
```

If you look at the default `logback.xml` in the spring-boot jar you will see that it uses some useful System properties which the `LoggingSystem` takes care of creating for you. These are:

- `${PID}` the current process ID.
- `${LOG_FILE}` if `logging.file` was set in Boot's external configuration.
- `${LOG_PATH}` if `logging.path` was set (representing a directory for log files to live in).

Spring Boot also provides some nice ANSI colour terminal output on a console (but not in a log file) using a custom Logback converter. See the default `base.xml` configuration for details.

If Groovy is on the classpath you should be able to configure Logback with `logback.groovy` as well (it will be given preference if present).

57.2 Configure Log4j for logging

Spring Boot supports Log4j for logging configuration, but it has to be on the classpath. If you are using the starter poms for assembling dependencies that means you have to exclude logback and then include log4j instead. If you aren't using the starter poms then you need to provide `commons-logging` (at least) in addition to Log4j.

The simplest path to using Log4j 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>${project.groupId}</groupId>
                    <artifactId>spring-boot-starter-logging</artifactId>
            </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j</artifactId>
</dependency>
```

Note

The use of the log4j starter gathers together the dependencies for common logging requirements (e.g. including having Tomcat use `java.util.logging` but configure the output using Log4j). See the Actuator Log4j Sample for more detail and to see it in action.

58. Data Access

58.1 Configure a DataSource

To override the default settings just define a `@Bean` of your own of type `DataSource`. See *Section 26.1, "Configure a DataSource"* in the "Spring Boot features" section and the `DataSourceAutoConfiguration` class for more details.

58.2 Use Spring Data repositories

Spring Data can create implementations for you of `@Repository` interfaces of various flavours. 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).

58.3 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 {

    //...

}
```

58.4 Configure JPA properties

Spring Data JPA already provides some vendor-independent configuration options (e.g. for SQL logging) and Spring Boot exposes those, and a few more for hibernate as external configuration properties. The most common options to set are:

```
spring.jpa.hibernate.ddl-auto: create-drop
spring.jpa.hibernate.naming_strategy: org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.database: H2
spring.jpa.show-sql: true
```

(Because of relaxed data binding hyphens or underscores should work equally well as property keys.) The `ddl-auto` setting is a special case in that it has different defaults depending on whether you are using an embedded database (`create-drop`) or not (`none`). In addition all properties in `spring.jpa.properties.*` are passed through as normal JPA properties (with the prefix stripped) when the local `EntityManagerFactory` is created.

See `HibernateJpaAutoConfiguration` and `JpaBaseConfiguration` for more details.

58.5 Use a custom EntityManagerFactory

To take full control of the configuration of the `EntityManagerFactory`, you need to add a `@Bean` named "entityManagerFactory". To avoid eager initialization of JPA infrastructure, Spring Boot auto-configuration does not switch on its entity manager based on the presence of a bean of that type. Instead it has to do it by name.

58.6 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.

59. 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.

59.1 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.

59.2 Initialize a database using Hibernate

You can set `spring.jpa.hibernate.ddl-auto` explicitly and the standard Hibernate property values are `none`, `validate`, `update`, `create-drop`. Spring Boot chooses a default value for you based on whether it thinks your database is embedded (default `create-drop`) or not (default `none`). An embedded database is detected by looking at the `Connection` type: `hsqldb`, `h2` and `derby` are embedded, the rest are not. Be careful when switching from in-memory to a “real” database that you don’t make assumptions about the existence of the tables and data in the new platform. You either have to set `ddl-auto` explicitly, or use one of the other mechanisms to initialize the database.

In addition, a file named `import.sql` in the root of the classpath will be executed on startup. This can be useful for demos and for testing if you are careful, but probably not something you want to be on the classpath in production. It is a Hibernate feature (nothing to do with Spring).

59.3 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 a file `schema-${platform}.sql` where `platform` is the vendor name of the database (`hsqldb`, `h2`, `oracle`, `mysql`, `postgresql` etc.). Spring Boot enables the failfast feature of the Spring JDBC initializer by default, so if the scripts cause exceptions the application will fail to start.

To disable the failfast you can set `spring.datasource.continueOnError=true`. This can be useful once an application has matured and been deployed a few times, since the scripts can act as “poor man’s migrations” — inserts that fail mean that the data is already there, so there would be no need to prevent the application from running, for instance.

59.4 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`.

59.5 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.

60. Batch applications

60.1 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.

61. Actuator

61.1 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 33.3, "Customizing the management server port"* in the "Production-ready features" section.

61.2 Customize the "whitelabel" error page

The Actuator installs a "whitelabel" error page that you will see in browser client if you encounter a server error (machine clients consuming JSON and other media types should see a sensible response with the right error code). To switch it off you can set `error.whitelabel.enabled=false`, but normally in addition or alternatively to that you will want to add your own error page replacing the whitelabel one. If you are using Thymeleaf you can do this by adding an `error.html` 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.

62. Security

62.1 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.

62.2 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.

62.3 Enable HTTPS

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 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_https` to `true` in `application.properties`.

63. Hot swapping

63.1 Reload static content

There are several options for hot reloading. Running in an IDE (especially with debugging on) is a good way to do development (all modern IDEs allow reloading of static resources and usually also hot-swapping of Java class changes). The Maven and Gradle plugins also support running from the command line with reloading of static files. You can use that with an external css/js compiler process if you are writing that code with higher level tools.

63.2 Reload Thymeleaf templates without restarting the container

If you are using Thymeleaf, then set `spring.thymeleaf.cache` to `false`. See `ThymeleafAutoConfiguration` for other template customization options.

63.3 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).

64. Build

64.1 Build an executable archive with Ant

To build with Ant you need to grab dependencies, compile and then create a jar or war archive as normal. To make it executable:

1. Use the appropriate launcher as a `Main-Class`, e.g. `JarLauncher` for a jar file, and specify the other properties it needs as manifest entries, principally a `Start-Class`.
2. Add the runtime dependencies in a nested "lib" directory (for a jar) and the `provided` (embedded container) dependencies in a nested `lib-provided` directory. Remember **not** to compress the entries in the archive.
3. Add the `spring-boot-loader` classes at the root of the archive (so the `Main-Class` is available).

Example:

```
<target name="build" depends="compile">
    <copy todir="target/classes/lib">
        <fileset dir="lib/runtime" />
    </copy>
    <jar destfile="target/spring-boot-sample-actuator-${spring-boot.version}.jar" compress="false">
        <fileset dir="target/classes" />
        <fileset dir="src/main/resources" />
        <zipfileset src="lib/loader/spring-boot-loader-jar-${spring-boot.version}.jar" />
        <manifest>
            <attribute name="Main-Class" value="org.springframework.boot.loader.JarLauncher" />
            <attribute name="Start-Class" value="${start-class}" />
        </manifest>
    </jar>
</target>
```

The Actuator Sample has a `build.xml` that should work if you run it with

```
$ ant -lib <path_to>/ivy-2.2.jar
```

after which you can run the application with

```
$ java -jar target/*.jar
```

65. Traditional deployment

65.1 Create a deployable war file

Use the `SpringBootServletInitializer` base class, which is picked up by Spring's Servlet 3.0 support on deployment. Add an extension of that to your project and build a war file as normal. For more detail, see the "Converting a jar Project to a war" guide on the spring.io website and the sample below.

The war file can also be executable if you use the Spring Boot build tools. In that case the embedded container classes (to launch Tomcat for instance) have to be added to the war in a `lib-provided` directory. The tools will take care of that as long as the dependencies are marked as "provided" in Maven or Gradle. Here's a Maven example in the Boot Samples.

65.2 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`.

65.3 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) {
        return application.sources(Application.class);
    }

}
```

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.

Part X. Appendices

Appendix A. Common application properties

Various properties can be specified inside your `application.properties`/`application.yml` file or as command line switches. This section provides a list common Spring Boot properties and references to the underlying classes that consume them.

> **Note**
>
> Property contributions can come from additional jar files on your classpath so you should not consider this an exhaustive list. It is also perfectly legit to define your own properties.

> **Warning**
>
> This sample file is meant as a guide only. Do **not** copy/paste the entire content into your application; rather pick only the properties that you need.

```
# ===================================================================
# COMMON SPRING BOOT PROPERTIES
#
# This sample file is provided as a guideline. Do NOT copy it in its
# entirety to your own application.               ^^^
# ===================================================================

# ----------------------------------------
# CORE PROPERTIES
# ----------------------------------------

# SPRING CONFIG (ConfigFileApplicationListener)
spring.config.name= # config file name (default to 'application')
spring.config.location= # location of config file

# PROFILES
spring.profiles= # comma list of active profiles

# APPLICATION SETTINGS (SpringApplication)
spring.main.sources=
spring.main.web-environment= # detect by default
spring.main.show-banner=true
spring.main....= # see class for all properties

# LOGGING
logging.path=/var/logs
logging.file=myapp.log
logging.config=

# IDENTITY (ContextIdApplicationContextInitializer)
spring.application.name=
spring.application.index=

# EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.port=8080
server.address= # bind to a specific NIC
server.session-timeout= # session timeout in sections
server.context-path= # the context path, defaults to '/'
server.servlet-path= # the servlet path, defaults to '/'
server.tomcat.access-log-pattern= # log pattern of the access log
server.tomcat.access-log-enabled=false # is access logging enabled
server.tomcat.protocol-header=x-forwarded-proto # ssl forward headers
server.tomcat.remote-ip-header=x-forwarded-for
server.tomcat.basedir=/tmp # base dir (usually not needed, defaults to tmp)
```

```
server.tomcat.background-processor-delay=30; # in seconds
server.tomcat.max-threads = 0 # number of threads in protocol handler
server.tomcat.uri-encoding = UTF-8 # character encoding to use for URL decoding

# SPRING MVC (HttpMapperProperties)
http.mappers.json-pretty-print=false # pretty print JSON
http.mappers.json-sort-keys=false # sort keys
spring.view.prefix= # MVC view prefix
spring.view.suffix= # ... and suffix
spring.resources.cache-period= # cache timeouts in headers sent to browser

# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.mode=HTML5
spring.thymeleaf.encoding=UTF-8
spring.thymeleaf.cache=true # set to false for hot refresh

# INTERNATIONALIZATION (MessageSourceAutoConfiguration)
spring.messages.basename=messages
spring.messages.encoding=UTF-8
spring.messages.cacheSeconds=-1


# SECURITY (SecurityProperties)
security.user.name=user # login username
security.user.password= # login password
security.user.role=USER # role assigned to the user
security.require-ssl=false # advanced settings ...
security.enable-csrf=false
security.basic.enabled=true
security.basic.realm=Spring
security.basic.path= # /**
security.headers.xss=false
security.headers.cache=false
security.headers.frame=false
security.headers.contentType=false
security.headers.hsts=all # none / domain / all
security.sessions=stateless # always / never / if_required / stateless
security.ignored=false

# DATASOURCE (DataSourceAutoConfiguration & AbstractDataSourceConfiguration)
spring.datasource.name= # name of the data source
spring.datasource.intialize=true # populate using data.sql
spring.datasource.schema= # a schema resource reference
spring.datasource.continueOnError=false # continue even if can't be initialized
spring.datasource.driverClassName= # JDBC Settings...
spring.datasource.url=
spring.datasource.username=
spring.datasource.password=
spring.datasource.max-active=100 # Advanced configuration...
spring.datasource.max-idle=8
spring.datasource.min-idle=8
spring.datasource.initial-size=10
spring.datasource.validation-query=
spring.datasource.test-on-borrow=false
spring.datasource.test-on-return=false
spring.datasource.test-while-idle=
spring.datasource.time-between-eviction-runs-millis=
spring.datasource.min-evictable-idle-time-millis=
spring.datasource.max-wait-millis=

# MONGODB (MongoProperties)
spring.data.mongodb.host= # the db host
spring.data.mongodb.port=27017 # the connection port (defaults to 27107)
spring.data.mongodb.uri=mongodb://localhost/test # connection URL

# JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.jpa.properties.*= # properties to set on the JPA connection
spring.jpa.openInView=true
spring.jpa.show-sql=true
```

```
spring.jpa.database-platform=
spring.jpa.database=
spring.jpa.generate-ddl=
spring.jpa.hibernate.naming-strategy= # naming classname
spring.jpa.hibernate.ddl-auto= # defaults to create-drop for embedded dbs

# JMX
spring.jmx.enabled=true # Expose MBeans from Spring

# RABBIT (RabbitProperties)
spring.rabbitmq.host= # connection host
spring.rabbitmq.port= # connection port
spring.rabbitmq.addresses= # connection addresses (e.g. myhost:9999,otherhost:1111)
spring.rabbitmq.username= # login user
spring.rabbitmq.password= # login password
spring.rabbitmq.virtualhost=
spring.rabbitmq.dynamic=


# REDIS (RedisProperties)
spring.redis.host=localhost # server host
spring.redis.password= # server password
spring.redis.port=6379 # connection port
spring.redis.pool.max-idle=8 # pool settings ...
spring.redis.pool.min-idle=0
spring.redis.pool.max-active=8
spring.redis.pool.max-wait=-1

# ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url=tcp://localhost:61616 # connection URL
spring.activemq.user=
spring.activemq.password=
spring.activemq.in-memory=true
spring.activemq.pooled=false

# JMS (JmsTemplateProperties)
spring.jms.pub-sub-domain=

# SPRING BATCH (BatchDatabaseInitializer)
spring.batch.job.names=job1,job2
spring.batch.job.enabled=true
spring.batch.initializer.enabled=true
spring.batch.schema= # batch schema to load

# AOP
spring.aop.auto=
spring.aop.proxyTargetClass=

# FILE ENCODING (FileEncodingApplicationListener)
spring.mandatory-file-encoding=false

# ----------------------------------------
# ACTUATOR PROPERTIES
# ----------------------------------------

# MANAGEMENT HTTP SERVER (ManagementServerProperties)
management.port= # defaults to 'server.port'
management.address= # bind to a specific NIC
management.contextPath= # default to '/'

# ENDPOINTS (AbstractEndpoint subclasses)
endpoints.autoconfig.id=autoconfig
endpoints.autoconfig.sensitive=true
endpoints.autoconfig.enabled=true
endpoints.beans.id=beans
endpoints.beans.sensitive=true
endpoints.beans.enabled=true
endpoints.configprops.id=configprops
endpoints.configprops.sensitive=true
endpoints.configprops.enabled=true
endpoints.configprops.keys-to-sanitize=password,secret
```

```
endpoints.dump.id=dump
endpoints.dump.sensitive=true
endpoints.dump.enabled=true
endpoints.env.id=env
endpoints.env.sensitive=true
endpoints.env.enabled=true
endpoints.health.id=health
endpoints.health.sensitive=false
endpoints.health.enabled=true
endpoints.info.id=info
endpoints.info.sensitive=false
endpoints.info.enabled=true
endpoints.metrics.id=metrics
endpoints.metrics.sensitive=true
endpoints.metrics.enabled=true
endpoints.shutdown.id=shutdown
endpoints.shutdown.sensitive=true
endpoints.shutdown.enabled=false
endpoints.trace.id=trace
endpoints.trace.sensitive=true
endpoints.trace.enabled=true

# MVC ONLY ENDPOINTS
endpoints.jolokia.path=jolokia
endpoints.jolokia.sensitive=true
endpoints.jolokia.enabled=true # when using Jolokia
endpoints.error.path=/error

# JMX ENDPOINT (EndpointMBeanExportProperties)
endpoints.jmx.enabled=true
endpoints.jmx.domain= # the JMX domain, defaults to 'org.springboot'
endpoints.jmx.unique-names=false
endpoints.jmx.enabled=true
endpoints.jmx.staticNames=

# JOLOKIA (JolokiaProperties)
jolokia.config.*= # See Jolokia manual

# REMOTE SHELL
shell.auth=simple # jaas, key, simple, spring
shell.command-refresh-interval=-1
shell.command-path-pattern= # classpath*:/commands/**, classpath*:/crash/commands/**
shell.config-path-patterns= # classpath*:/crash/*
shell.disabled-plugins=false # don't expose plugins
shell.ssh.enabled= # ssh settings ...
shell.ssh.keyPath=
shell.ssh.port=
shell.telnet.enabled= # telnet settings ...
shell.telnet.port=
shell.auth.jaas.domain= # authentication settings ...
shell.auth.key.path=
shell.auth.simple.user.name=
shell.auth.simple.user.password=
shell.auth.spring.roles=

# GIT INFO
spring.git.properties= # resource ref to generated git info properties file
```

Appendix B. 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).

B.1 From the “spring-boot-autoconfigure” module

The following auto-configuration classes are from the `spring-boot-autoconfigure` module:

Configuration Class	Links
AopAutoConfiguration	javadoc
BatchAutoConfiguration	javadoc
DataSourceAutoConfiguration	javadoc
DataSourceTransactionManagerAutoConfiguration	javadoc
DeviceResolverAutoConfiguration	javadoc
DispatcherServletAutoConfiguration	javadoc
EmbeddedServletContainerAutoConfiguration	javadoc
HibernateJpaAutoConfiguration	javadoc
HttpMessageConvertersAutoConfiguration	javadoc
JmsTemplateAutoConfiguration	javadoc
JmxAutoConfiguration	javadoc
JpaRepositoriesAutoConfiguration	javadoc
MessageSourceAutoConfiguration	javadoc
MongoAutoConfiguration	javadoc
MongoRepositoriesAutoConfiguration	javadoc
MongoTemplateAutoConfiguration	javadoc
MultipartAutoConfiguration	javadoc
PropertyPlaceholderAutoConfiguration	javadoc
RabbitAutoConfiguration	javadoc
ReactorAutoConfiguration	javadoc
RedisAutoConfiguration	javadoc
SecurityAutoConfiguration	javadoc

Configuration Class	Links
ServerPropertiesAutoConfiguration	javadoc
ThymeleafAutoConfiguration	javadoc
WebMvcAutoConfiguration	javadoc
WebSocketAutoConfiguration	javadoc

B.2 From the “spring-boot-actuator” module

The following auto-configuration classes are from the `spring-boot-actuator` module:

Configuration Class	Links
AuditAutoConfiguration	javadoc
CrshAutoConfiguration	javadoc
EndpointAutoConfiguration	javadoc
EndpointMBeanExportAutoConfiguration	javadoc
EndpointWebMvcAutoConfiguration	javadoc
ErrorMvcAutoConfiguration	javadoc
JolokiaAutoConfiguration	javadoc
ManagementSecurityAutoConfiguration	javadoc
ManagementServerPropertiesAutoConfiguration	javadoc
MetricFilterAutoConfiguration	javadoc
MetricRepositoryAutoConfiguration	javadoc
TraceRepositoryAutoConfiguration	javadoc
TraceWebFilterAutoConfiguration	javadoc

Appendix C. 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.

C.1 Nested JARs

Java does not provide any standard way to load nested jar files (i.e. jar files that are themselves contained within a jar). This can be problematic if you are looking to distribute a self contained application that you can just run from the command line without unpacking.

To solve this problem, many developers use "shaded" jars. A shaded jar simply packages all classes, from all jars, into a single *uber jar*. The problem with shaded jars is that it becomes hard to see which libraries you are actually using in your application. It can also be problematic if the the same filename is used (but with different content) in multiple jars. Spring Boot takes a different approach and allows you to actually nest jars directly.

The executable jar file structure

Spring Boot Loader compatible jar files should be structured in the following way:

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-com
 |  +-mycompany
 |     + project
 |        +-YouClasses.class
 +-lib
    +-dependency1.jar
    +-dependency2.jar
```

Dependencies should be placed in a nested `lib` directory.

The executable war file structure

Spring Boot Loader compatible war files should be structured in the following way:

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
```

```
 |        +-loader
 |           +-<spring boot loader classes>
 +-WEB-INF
    +-classes
    |  +-com
    |     +-mycompany
    |        +-project
    |           +-YouClasses.class
    +-lib
    |  +-dependency1.jar
    |  +-dependency2.jar
    +-lib-provided
       +-servlet-api.jar
       +-dependency3.jar
```

Dependencies should be placed in a nested `WEB-INF/lib` directory. Any dependencies that are required when running embedded but are not required when deploying to a traditional web container should be placed in `WEB-INF/lib-provided`.

C.2 Spring Boot's "JarFile" class

The core class used to support loading nested jars is `org.springframework.boot.loader.jar.JarFile`. It allows you load jar content from a standard jar file, or from nested child jar data. When first loaded, the location of each `JarEntry` is mapped to a physical file offset of the outer jar:

```
myapp.jar
+---------+---------------------+
|         | /lib/mylib.jar      |
| A.class |+---------+---------+|
|         || B.class | B.class ||
|         |+---------+---------+|
+---------+---------------------+
^          ^          ^
0063       3452       3980
```

The example above shows how `A.class` can be found in `myapp.jar` position `0063`. `B.class` from the nested jar can actually be found in `myapp.jar` position `3452` and `B.class` is at position `3980`.

Armed with this information, we can load specific nested entries by simply seeking to appropriate part if the outer jar. We don't need to unpack the archive and we don't need to read all entry data into memory.

Compatibility with the standard Java "JarFile"

Spring Boot Loader strives to remain compatible with existing code and libraries. `org.springframework.boot.loader.jar.JarFile` extends from `java.util.jar.JarFile` and should work as a drop-in replacement. The `RandomAccessJarFile.getURL()` method will return a `URL` that opens a `java.net.JarURLConnection` compatible connection. `RandomAccessJarFile` URLs can be used with Java's `URLClassLoader`.

C.3 Launching executable jars

The `org.springframework.boot.loader.Launcher` class is a special bootstrap class that is used as an executable jars main entry point. It is the actual `Main-Class` in your jar file and it's used to setup an appropriate `URLClassLoader` and ultimately call your `main()` method.

There are 3 launcher subclasses (`JarLauncher`, `WarLauncher` and `PropertiesLauncher`). Their purpose is to load resources (`.class` files etc.) from nested jar files or war files in directories (as opposed to explicitly on the classpath). In the case of the `[Jar|War]Launcher` the nested paths

are fixed (`lib/*.jar` and `lib-provided/*.jar` for the war case) so you just add extra jars in those locations if you want more. The `PropertiesLauncher` looks in `lib/` by default, but you can add additional locations by setting an environment variable `LOADER_PATH` or `loader.path` in `application.properties` (comma-separated list of directories or archives).

Launcher manifest

You need to specify an appropriate `Launcher` as the `Main-Class` attribute of `META-INF/MANIFEST.MF`. The actual class that you want to launch (i.e. the class that you wrote that contains a `main` method) should be specified in the `Start-Class` attribute.

For example, here is a typical `MANIFEST.MF` for an executable jar file:

```
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

For a war file, it would be:

```
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```

> **Note**
>
> You do not need to specify `Class-Path` entries in your manifest file, the classpath will be deduced from the nested jars.

Exploded archives

Certain PaaS implementations may choose to unpack archives before they run. For example, Cloud Foundry operates in this way. You can run an unpacked archive by simply starting the appropriate launcher:

```
$ unzip -q myapp.jar
$ java org.springframework.boot.loader.JarLauncher
```

C.4 PropertiesLauncher Features

`PropertiesLauncher` has a few special features that can be enabled with external properties (System properties, environment variables, manifest entries or `application.properties`).

Key	Purpose
`loader.path`	Comma-separated Classpath, e.g. `lib:${HOME}/app/lib`.
`loader.home`	Location of additional properties file, e.g. `file:///opt/app` (defaults to `${user.dir}`)
`loader.args`	Default arguments for the main method (space separated)
`loader.main`	Name of main class to launch, e.g. `com.app.Application`.
`loader.config.name`	Name of properties file, e.g. `loader` (defaults to `application`).
`loader.config.location`	Path to properties file, e.g. `classpath:loader.properties` (defaults to `application.properties`).

Key	Purpose
`loader.system`	Boolean flag to indicate that all properties should be added to System properties (defaults to `false`)

Manifest entry keys are formed by capitalizing initial letters of words and changing the separator to "`-`" from "`.`" (e.g. `Loader-Path`). The exception is `loader.main` which is looked up as `Start-Class` in the manifest for compatibility with `JarLauncher`).

Environment variables can be capitalized with underscore separators instead of periods.

- `loader.home` is the directory location of an additional properties file (overriding the default) as long as `loader.config.location` is not specified.
- `loader.path` can contain directories (scanned recursively for jar and zip files), archive paths, or wildcard patterns (for the default JVM behavior).
- Placeholder replacement is done from System and environment variables plus the properties file itself on all values before use.

C.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.

C.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