

## 4.4. Developing Your First Spring Boot Application

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

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

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

### 4.4.1. Prerequisites

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

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

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

#### Maven

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

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

#### Gradle

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

```
$ gradle --version
```

```
-----  
Gradle 8.1.1  
-----
```

```
Build time: 2023-04-21 12:31:26 UTC  
Revision: 1cf537a851c635c364a4214885f8b9798051175b  
  
Kotlin: 1.8.10  
Groovy: 3.0.15  
Ant: Apache Ant(TM) version 1.10.11 compiled on July 10 2021  
JVM: 17.0.7 (BellSoft 17.0.7+7-LTS)  
OS: Linux 6.2.12-200.fc37.aarch64 aarch64
```

#### 4.4.2. Setting up the project with Maven

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

```
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
      https://maven.apache.org/xsd/maven-4.0.0.xsd">  
  <modelVersion>4.0.0</modelVersion>  
  
  <groupId>com.example</groupId>  
  <artifactId>myproject</artifactId>  
  <version>0.0.1-SNAPSHOT</version>  
  
  <parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>3.2.5</version>  
  </parent>  
  
  <!-- Additional lines to be added here... -->  
  
</project>
```

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

#### NOTE

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

### 4.4.3. Setting up the project with Gradle

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

```
plugins {  
    id 'java'  
    id 'org.springframework.boot' version '3.2.5'  
}  
  
apply plugin: 'io.spring.dependency-management'  
  
group = 'com.example'  
version = '0.0.1-SNAPSHOT'  
sourceCompatibility = '17'  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
}
```

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

#### NOTE

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

### 4.4.4. Adding Classpath Dependencies

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

#### Maven

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

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

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

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

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

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

## Gradle

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

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

```
$ gradle dependencies

> Task :dependencies

-----
Root project 'myproject'
-----
```

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

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

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

### 4.4.5. Writing the Code

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

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

### Java

```
package com.example;

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

@RestController
@SpringBootApplication
public class MyApplication {

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

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

}
```

### Kotlin

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

@RestController
@SpringBootApplication
class MyApplication {

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

}

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

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

## The @RestController and @RequestMapping Annotations

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

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

### TIP

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

## The @SpringBootApplication Annotation

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

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

### Starters and Auto-configuration

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

## The “main” Method

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

## 4.4.6. Running the Example

### Maven

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

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

```
.  
/\ \ / ____' _ __ _(_)_ __   __ _ \ \ \ \ \  
( ( )\___| '_ \|'_|| |'_ \| \ \ \ \ \  
\ \ / ___)| |_) | | | | | | | (| | ) ) ) )  
' |____| .__|_| |_| | |_\__, | / / / /  
=====|_|=====|___/=/_/_/_/  
:: Spring Boot :: (v3.2.5)  
  
.....  
..... (log output here)  
.....  
..... Started MyApplication in 0.906 seconds (process running for 6.514)
```

```
Hello World!
```

## Gradle

```
$ gradle bootRun

      .
     /\ /  ____' _ ____ _ ( ) _ __ _ _ _ \ \ \ \ \
  ( ( ) \ ___ | ' _ | ' _ | | ' _ \ \ \ \ \ \ \ \ \
   \ \  ___ ) | | _ | | | | | | | ( _ | | ) ) ) )
    '  | ___ | . _ | | | _ | | \ __, | / / / / /
   =====|_|=====|___/=/_/_/_/
:: Spring Boot :: (v3.2.5)

.....
..... (log output here)
.....
..... Started MyApplication in 0.906 seconds (process running for 6.514)
```

```
Hello World!
```

### 4.4.7. Creating an Executable Jar

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

#### Executable jars and Java

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

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

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

#### Maven

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

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

#### NOTE

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

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



```
$ mvn package
```

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building myproject 0.0.1-SNAPSHOT
[INFO] -----
[INFO] .... ..
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ myproject ---
[INFO] Building jar: /Users/developer/example/spring-boot-example/target/myproject-
0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:3.2.5:repackage (default) @ myproject ---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

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

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

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

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

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

```
  .
 / \ / ___ ' _ _ _ _ ( _ ) _ _ _ _ \ \ \ \
( ( ) \ ___ | ' _ | ' _ | ' _ \ _ ' \ \ \ \
 \ \ ___ | | _ | | | | | | ( _ | ) ) ) )
  ' | ___ | . _ | | _ | | \ _ , | / / / /
=====|_|=====|___/=/_/_/_/_/
:: Spring Boot :: (v3.2.5)
.....
..... . . . (log output here)
.....
..... Started MyApplication in 0.999 seconds (process running for 1.253)
```

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

## Gradle

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

```
$ gradle bootJar
```

```
BUILD SUCCESSFUL in 639ms
```

```
3 actionable tasks: 3 executed
```

```
$ gradle bootJar
```

```
BUILD SUCCESSFUL in 639ms
```

```
3 actionable tasks: 3 executed
```

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

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

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

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

      .
     /\ /  ____' _ ____ _ ( _ ) _ ____ _ _ \ \ \ \
  ( ( ) \ ____ | ' _ | ' _ | | ' _ \ \ _ | \ \ \ \
    \ \ ____ | | _ | | | | | | | ( _ | | ) ) ) )
      ' | ____ | . _ | | | _ | | \ __, | / / / /
  =====|_|=====| ____ / = / _ / _ /
  :: Spring Boot :: (v3.2.5)

.....
..... (log output here)
.....
..... Started MyApplication in 0.999 seconds (process running for 1.253)
```

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

  .
 / \ /  _ _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \ \
( ( ) \ _ _ | ' _ | ' _ | ' _ \ \ _ | \ \ \ \ \
 \ \ _ _ | | _ | | | | | | | ( _ | | ) ) ) )
  ' | _ _ | . _ _ | | _ | | _ \ _ _ | / / / /
=====|_|=====| _ _ / _ / _ / _ /
:: Spring Boot :: (v3.2.5)

.....
..... (log output here)
.....
..... Started MyApplication in 0.999 seconds (process running for 1.253)
```

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

## 4.5. What to Read Next

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

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

# Chapter 5. Upgrading Spring Boot

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

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

## 5.1. Upgrading From 1.x

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

## 5.2. Upgrading to a New Feature Release

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

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

### WARNING

Properties that are added late to the environment, such as when using `@PropertySource`, will not be taken into account.

### NOTE

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

## 5.3. Upgrading the Spring Boot CLI

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

## 5.4. What to Read Next

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