Spring Boot Reference Documentation

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

Table of Contents

2.4.3

Legal

Chapter 1. Spring Boot Documentation

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

1.1. About the Documentation

The Spring Boot reference guide is available as:

- Multi-page HTML
- Single page HTML
- PDF

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

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

1.2. Getting Help

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

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

> ℹ All of Spring Boot is open source, including the documentation. If you find problems with the docs or if you want to improve them, please get involved.

1.3. Upgrading from an Earlier Version

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.

You should always ensure that you are running a supported version of Spring Boot.

1.4. First Steps

If you are getting started with Spring Boot or 'Spring' in general, start with the following topics:

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

1.5. Working with Spring Boot

Ready to actually start using Spring Boot? We have you covered:

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

1.6. Learning about Spring Boot Features

Need more details about Spring Boot's core features? The following content is for you:

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

1.7. Moving to Production

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

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

1.8. Advanced Topics

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

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

Chapter 2. Getting Started

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

2.1. Introducing Spring Boot

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

You can use Spring Boot to create Java applications that can be started by using `java -jar` or more traditional war deployments. 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 (such as embedded servers, security, metrics, health checks, and externalized configuration).
- Absolutely no code generation and no requirement for XML configuration.

2.2. System Requirements

Spring Boot 2.4.3 requires Java 8 and is compatible up to Java 15 (included). Spring Framework 5.3.4 or above is also required.

Explicit build support is provided for the following build tools:

Build Tool	Version
Maven	3.3+
Gradle	6 (6.3 or later). 5.6.x is also supported but in a deprecated form

2.2.1. Servlet Containers

Spring Boot supports the following embedded servlet containers:

Name	Servlet Version
Tomcat 9.0	4.0

Name	Servlet Version
Jetty 9.4	3.1
Undertow 2.0	4.0

You can also deploy Spring Boot applications to any Servlet 3.1+ compatible container.

2.3. Installing Spring Boot

Spring Boot can be used with “classic” Java development tools or installed as a command line tool. Either way, you need Java SDK v1.8 or higher. Before you begin, you should check your current Java installation by using the following command:

```
$ java -version
```

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

2.3.1. Installation Instructions for the Java Developer

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

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

Maven Installation

Spring Boot is compatible with Apache Maven 3.3 or above. If you do not already have Maven installed, you can follow the instructions at maven.apache.org.

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

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

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

Gradle Installation

Spring Boot is compatible with Gradle 6 (6.3 or later). Gradle 5.6.x is also supported but this support is deprecated and will be removed in a future release. If you do not already have Gradle installed, you can follow the instructions at gradle.org.

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

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

More details on getting started with Spring Boot and Gradle can be found in the Getting Started section of the Gradle plugin's reference guide.

2.3.2. Installing the Spring Boot CLI

The Spring Boot CLI (Command Line Interface) is a command line tool that you can use to quickly prototype with Spring. It lets you run Groovy scripts, which means that you have a familiar Java-like syntax without so much boilerplate code.

You do not need to use the CLI to work with Spring Boot, but it is 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-2.4.3-bin.zip
- spring-boot-cli-2.4.3-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. Alternatively, you can use `java -jar` with the `.jar` file (the script helps you to be sure that the classpath is set correctly).

Installation with SDKMAN!

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

```
$ sdk install springboot
$ spring --version
Spring Boot v2.4.3
```

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

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-
cli-2.4.3-bin/spring-2.4.3/
$ sdk default springboot dev
$ spring --version
Spring CLI v2.4.3
```

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

You can see it by running the following command:

```
$ sdk ls springboot

================================================================================
Available Springboot Versions
================================================================================
> + dev
* 2.4.3

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

OSX Homebrew Installation

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

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

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

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

MacPorts Installation

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

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

Command-line Completion

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

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

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

Windows Scoop Installation

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

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

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

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

Quick-start Spring CLI Example

You can use the following web application to test your installation. To start, create a file called `app.groovy`, as follows:

```
@RestController
class ThisWillActuallyRun {

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

}
```

Then run it from a shell, as follows:

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

> The first run of your application is slow, as dependencies are downloaded. Subsequent runs are much quicker.

Open `localhost:8080` in your favorite web browser. You should see the following output:

```
Hello World!
```

2.3.3. Upgrading from an Earlier Version of Spring Boot

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.

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>
```

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

> Once you're done with the migration, please make sure to remove this module from your project's dependencies.

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.

2.4. 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. We use Maven to build this project, since most IDEs support it.

> The spring.io web site 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 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 Spring Initializr documentation for more details.

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

```
$ java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)
```

```
$ mvn -v
Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06-17T14:33:14-04:00)
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java version: 1.8.0_102, vendor: Oracle Corporation
```

> 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.

2.4.1. Creating the POM

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>2.4.3</version>
    </parent>

    <description/>
    <developers>
        <developer/>
    </developers>
    <licenses>
        <license/>
    </licenses>
    <scm>
        <url/>
    </scm>
    <url/>

    <!-- 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).

> ℹ 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.

2.4.2. Adding Classpath Dependencies

Spring Boot provides a number of “Starters” that let you add jars to your classpath. Our applications for smoke tests 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.

Other “Starters” provide dependencies that you are likely to need when developing a specific type of application. 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.

2.4.3. Writing the Code

To finish our application, we need to create a single Java file. By default, Maven compiles sources from `src/main/java`, so you need to create that directory structure and then add a file named `src/main/java/Example.java` to contain the following code:

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

@RestController
@EnableAutoConfiguration
public class Example {

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

    public static void main(String[] args) {
        SpringApplication.run(Example.class, 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 `Example` class is `@RestController`. This is known as a *stereotype* annotation. It provides hints for people reading the code and for Spring that the class plays a specific role. In this case, our class is a web `@Controller`, so Spring 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.

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

The @EnableAutoConfiguration Annotation

The second class-level annotation is `@EnableAutoConfiguration`. This annotation tells Spring Boot to “guess” how you 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 `Example.class` as an argument to the `run` method to tell `SpringApplication` which is the primary Spring component. The `args` array is also passed through to expose any command-line arguments.

2.4.4. Running the Example

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 ::  (v2.4.3)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.514)
```

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

```
Hello World!
```

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

2.4.5. 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 “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.

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>
```

ℹ 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:2.4.3: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`, 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 ::  (v2.4.3)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.536 seconds (JVM running for 2.864)
```

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

2.5. What to Read Next

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

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

Chapter 3. Using Spring Boot

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

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

3.1. Build Systems

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

3.1.1. Dependency Management

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

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

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

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

3.1.2. Maven

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

- Reference (HTML and PDF)
- API

3.1.3. Gradle

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

- Reference (HTML and PDF)
- API

3.1.4. Ant

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

To declare dependencies, a typical `ivy.xml` file looks something like the following example:

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

A typical `build.xml` looks like the following example:

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

    <property name="spring-boot.version" value="2.4.3" />

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

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

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

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

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

If you do not want to use the `spring-boot-antlib` module, see the *Build an Executable Archive from Ant without Using spring-boot-antlib* “How-to” .

3.1.5. Starters

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

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

What’s in a name

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

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

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

Table 1. Spring Boot application starters

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

Name	Description
spring-boot-starter-data-ldap	Starter for using Spring Data LDAP
spring-boot-starter-data-mongodb	Starter for using MongoDB document-oriented database and Spring Data MongoDB
spring-boot-starter-data-mongodb-reactive	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive
spring-boot-starter-data-neo4j	Starter for using Neo4j graph database and Spring Data Neo4j
spring-boot-starter-data-r2dbc	Starter for using Spring Data R2DBC
spring-boot-starter-data-redis	Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client
spring-boot-starter-data-redis-reactive	Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client
spring-boot-starter-data-rest	Starter for exposing Spring Data repositories over REST using Spring Data REST
spring-boot-starter-data-solr	Starter for using the Apache Solr search platform with Spring Data Solr. Deprecated since 2.3.9
spring-boot-starter-freemarker	Starter for building MVC web applications using FreeMarker views
spring-boot-starter-groovy-templates	Starter for building MVC web applications using Groovy Templates views
spring-boot-starter-hateoas	Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS
spring-boot-starter-integration	Starter for using Spring Integration
spring-boot-starter-jdbc	Starter for using JDBC with the HikariCP connection pool
spring-boot-starter-jersey	Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web
spring-boot-starter-jooq	Starter for using jOOQ to access SQL databases. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc
spring-boot-starter-json	Starter for reading and writing json
spring-boot-starter-jta-atomikos	Starter for JTA transactions using Atomikos
spring-boot-starter-jta-bitronix	Starter for JTA transactions using Bitronix. Deprecated since 2.3.0
spring-boot-starter-mail	Starter for using Java Mail and Spring Framework’s email sending support

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

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

Table 2. Spring Boot production starters

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

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

Table 3. Spring Boot technical starters

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

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

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

3.2. Structuring Your Code

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

3.2.1. Using the “default” Package

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

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

3.2.2. Locating the Main Application Class

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

> If you don’t want to use @SpringBootApplication, the @EnableAutoConfiguration and @ComponentScan annotations that it imports defines that behaviour so you can also use those instead.

The following listing shows a typical layout:

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

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

```
package com.example.myapplication;

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

@SpringBootApplication
public class Application {

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

}
```

3.3. Configuration Classes

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

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

3.3.1. Importing Additional Configuration Classes

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

3.3.2. Importing XML Configuration

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

3.4. Auto-configuration

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

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

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

3.4.1. Gradually Replacing Auto-configuration

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

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

3.4.2. Disabling Specific Auto-configuration Classes

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

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

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

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

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

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

3.5. Spring Beans and Dependency Injection

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. We often find that using `@ComponentScan` (to find your beans) and using `@Autowired` (to do 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.) are automatically registered as Spring Beans.

The following example shows a `@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;
    }

    // ...

}
```

If a bean has one constructor, you can omit the `@Autowired`, as shown in the following example:

```
@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...

}
```

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

3.6. Using the @SpringBootApplication Annotation

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

- `@EnableAutoConfiguration`: enable Spring Boot’s auto-configuration mechanism
- `@ComponentScan`: enable `@Component` scan on the package where the application is located (see the best practices)

- `@Configuration`: allow to register extra beans in the context or import additional configuration classes

```
package com.example.myapplication;

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

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

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

}
```

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

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

```
package com.example.myapplication;

import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration(proxyBeanMethods = false)
@EnableAutoConfiguration
@Import({ MyConfig.class, MyAnotherConfig.class })
public class Application {

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

}
```

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