Developing With Spring Boot - Spring Boot 4.0.1

Compiled by: Benjamin Soto-Roberts

Created: 01/06/26

Aligned to: Spring Boot 4.0.1 Reference Documentation; see Resources for links.

Disclaimer: This document is a curated summary and does not replace the official Spring Boot documentation.

What is Spring Boot?

Spring Boot helps you to create stand-alone, production-grade Spring-based applications that you can run. Spring Boot takes 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. (Spring, n.d.-a)

The 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 (when not targeting native image) and no requirement for XML configuration.

What is IoC?

“In software design, **inversion of control** (**IoC**) is a design principle in which custom-written portions of a computer program receive the flow of control from an external source (e.g. a framework).” (Wikipedia, 2025)

Dependency injection (DI) is a specialized form of IoC, whereby objects define their dependencies (that is, the other objects they work with) only through constructor

arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. The IoC container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes or a mechanism such as the Service Locator pattern. (Spring, n.d.-j)

Spring Boot Best Practices

Build Systems

It is strongly recommended to choose a build system that supports dependency management and that can consume artifacts published to the Maven Central Repository. (Spring, n.d.-b)

- Recommended Build Systems
    - Maven
    - Gradle

Dependency Management – Each release of Spring Boot provides a curated list of dependencies that it supports.

- No version required in build configuration – Spring manages versions
- Automatically upgraded with Spring

Starters – Starters are a set of convenient dependency descriptors that you can include in your application build. 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.

- All **official** starters follow a similar naming pattern; spring-boot-starter-*, where * is a particular type of application.
- You can create your own starters. Third party starters should not start with spring-boot, as it is reserved for official Spring Boot artifacts

Structuring Your Code

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

Using the "default" Package – When a class does not include a package declaration, it is in the "default package". The use of the "default package" is generally discouraged and should be avoided. (Spring, n.d.-c)

- Since every class from every jar is read through automatic configuration and component scanning, if these scanning mechanisms don't have a clear "base package" to start from Spring Boot might try to scan every class in every JAR file on your classpath resulting in:
    - Slower Startup Time - Scanning tons of unnecessary classes takes a long time
    - Unexpected Beans/Entities - Spring might accidentally pick up classes from third-party libraries (JARs) that you didn't intend to be Spring components or JPA entities, leading to strange errors. (Runebook.dev, 2025)

Locating the Main Application Class – It is generally recommended to 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.
- Using a root package also allows component scan to apply only on your project.

The following listing shows a typical layout where the MyApplication.java file would declare the main method, along with the basic @SpringBootApplication.

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

Configuration Classes

Spring Boot favors Java-based configuration. It is possible to use SpringApplication with XML sources however, it is generally recommended that the primary source be a single @Configuration class. (Spring, n.d.-d)

- Usually, the class that defines the main method is a good candidate as the primary @Configuration.

Importing Additional Configuration Classes – Multiple configuration classes can exist using the @Configuration annotation

- The @Import annotation can be used to import additional configuration classes.
- Alternatively, @ComponentScan to automatically pick up all Spring components, including @Configuration classes.

Importing XML Configuration – If configuration must be XML based, it is still recommended to start with a @Configuration class.

- Use an @ImportResource annotation to load XML configuration files

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. (Spring, n.d.-e)

- Autoconfiguration is opt-in by adding the @EnableAutoConfiguration or the @SpringBootApplication annotations to one of your @Configuration classes.
- You should only ever add one @SpringBootApplication or @EnableAutoConfiguration annotation and is recommended to be your primary @Configuration class.

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.

- To find out what auto-configuration is currently being applied, and why, start your application with the --debug switch enabling debug logs for a selection of core loggers and logs a condition report to the console.

Disabling Specific Auto-configuration Classes – For auto-configuration classes that you do not want are being applied, you can use the exclude attribute of @SpringBootApplication to disable them.

- @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
- If the class is not on the classpath, you can use the excludeName attribute of the annotation and specify the fully qualified name instead.
- You can define exclusions both at the annotation level and by using the property.

Auto-configuration Packages - Auto-configuration packages are the packages that various auto-configured features look in by default when scanning for things such as entities and Spring Data repositories.

- The @EnableAutoConfiguration annotation (either directly or through its presence on @SpringBootApplication) determines the default auto-configuration package.
- Additional packages can be configured using the @AutoConfigurationPackage annotation.

Spring Beans and Dependency Injection

Although you are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies, it is generally recommend using constructor injection to wire up dependencies and @ComponentScan to find beans. (Spring, n.d.-f)

- If you structure your code as suggested (locating your application class in a top package), you can add @ComponentScan without any arguments or use the @SpringBootApplication annotation which implicitly includes it. All the application components (@Component, @Service, @Repository, @Controller, and others) are automatically registered as Spring Beans.
- If a bean has more than one constructor, you will need to mark the one you want Spring to use with @Autowired
- Using constructor injection lets the injected dependencies field be marked as final, indicating that it cannot be subsequently changed. – Thread safety

```
20  @Service
21  public class ProductService {
22
23      // Final for thread safety
24      private final ProductRepository repository;
25
26      /*The ProductRepository is injected into the constructor. SpringBoot automatically injects dependencies into a class
27       with a single constructor*/
28      public ProductService(ProductRepository repository) {
29          this.repository = repository;
30      }
31
```

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: (Spring, n.d.-g)

- @EnableAutoConfiguration: enable Spring Boot's auto-configuration mechanism
- @ComponentScan: enable @Component scan on the package where the application is located

- @SpringBootConfiguration: enable registration of extra beans in the context or the import of additional configuration classes. An alternative to Spring's standard @Configuration that aids configuration detection in your integration tests.

Running Your Application

One of the biggest advantages of packaging your application as a jar and using an embedded HTTP server is that you can run your application as you would any other. The same applies to debugging Spring Boot applications. You do not need any special IDE plugins or extensions. (Spring, n.d.-h)

Running From an IDE – A Spring Boot application can be run from an IDE as a Java application however, for the application to run the project must be imported.

- Import steps vary depending on the IDE and build system.
- Most IDEs can import Maven projects directly.
- Maven includes plugins for Eclipse and IntelliJ IDEA and Gradle offers plugins for various IDEs to be able to import a project direct from the IDE.

Running as a Packaged Application – If Spring Boot Maven or Gradle plugins are used to create an executable jar, the application can be run.

- Command Example: *java -jar target/myapplication-0.0.1-SNAPSHOT.jar*
- The jar will be packaged in the target directory of the Spring Boot project when using Maven.

Using the Maven Plugin – The Spring Boot Maven plugin includes a run goal that can be used to quickly compile and run the application. Applications run in an exploded form, as they do in the IDE.

- Command Example: *mvn spring-boot:run*

Using the Gradle Plugin – The Spring Boot Gradle plugin also includes a bootRun task that can be used to run your application in an exploded form. The bootRun task is added whenever you apply the org.springframework.boot and java plugins.

- Command Example: *gradle bootRun*

Developer Tools

Spring Boot includes an additional set of tools that can make the application development experience a little more pleasant. The spring-boot-devtools module can be included in any project to provide additional development-time features. (Spring, n.d.-i)

- To include devtools support, add the module dependency to your build

Maven

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

Gradle

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

- Devtools might cause classloading issues, particularly in multi-module projects
- Developer tools are automatically disabled when running a fully packaged application.
- Repackaged archives do not contain devtools by default.

Property Defaults – Several of the libraries supported by Spring Boot use caches to improve performance. While caching is very beneficial in production, it can be counter-productive during development, preventing you from seeing the changes you just made in your application. For this reason, spring-boot-devtools disables the caching options by default.

- Cache options are usually configured by settings in your application.properties file.
- Rather than needing to set these properties manually, the spring-boot-devtools module automatically applies sensible development-time configuration.
- If you do not want property defaults to be applied, you can set spring.devtools.add-properties to false in your application.properties.

While developing Spring MVC and Spring WebFlux applications, developer tools suggests enabling DEBUG logging for the web logging group. This will give you information about the incoming request, which handler is processing it, the response outcome, and other details.

- If you wish to log all request details (including potentially sensitive information), you can turn on the spring.mvc.log-request-details or spring.http.codecs.log-request-details configuration properties.

Automatic Restart – Applications that use spring-boot-devtools automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE, as it gives a very fast feedback loop for code changes.

- Restart functionality does not work well with objects that are deserialized by using a standard ObjectInputStream
- Unfortunately, several third-party libraries deserialize without considering the context classloader. If you find such a problem, you need to request a fix with the original authors.

Resources

Spring. (n.d.-a). *Spring Boot*.

https://docs.spring.io/spring-boot/index.html

Spring. (n.d.-b). *Build Systems*.

https://docs.spring.io/spring-boot/reference/using/build-systems.html

Spring. (n.d.-c). *Structuring Your Code*.

https://docs.spring.io/spring-boot/reference/using/structuring-your-code.html

Spring. (n.d.-d). *Configuration Classes*.

https://docs.spring.io/spring-boot/reference/using/configuration-classes.html

Spring. (n.d.-e). *Auto-configuration*.

https://docs.spring.io/spring-boot/reference/using/auto-configuration.html

Spring. (n.d.-f). *Spring Beans and Dependency Injection*.

https://docs.spring.io/spring-boot/reference/using/spring-beans-and-dependency-injection.html

Spring. (n.d.-g). *Using the @SpringBootApplication Annotation*.

https://docs.spring.io/spring-boot/reference/using/using-the-springbootapplication-annotation.html

Spring. (n.d.-h). *Running Your Application*.

https://docs.spring.io/spring-boot/reference/using/running-your-application.html

Spring. (n.d.-i). *Developer Tools*.

https://docs.spring.io/spring-boot/reference/using/devtools.html

Spring. (n.d.-j). *Introduction to the Spring IoC Container and Beans*.

https://docs.spring.io/spring-framework/reference/core/beans/introduction.html

Runebook.dev. (2025, Sept 27). *Spring Boot Default Package: Why It's a Bad Idea and the Simple Fix*. *https://runebook.dev/en/docs/spring_boot/using/using.structuring-your-code.using-the-default-package*

Wikipedia. (2025, Dec. 3). *Inversion of Control*.

https://en.wikipedia.org/wiki/Inversion_of_control