What is Spring Boot and What Are Its Main Features?

Spring Boot is essentially a framework for rapid application development built on top of the Spring Framework. With its auto-configuration and embedded application server support, combined with the extensive documentation and community support it enjoys, Spring Boot is one of the most popular technologies in the Java ecosystem as of date.

Here are a few salient features:

- **Starters** – a set of dependency descriptors to include relevant dependencies at a go
- **Auto-configuration** – a way to automatically configure an application based on the dependencies present on the classpath
- **Actuator** – to get production-ready features such as monitoring
- **Security**
- **Logging**

What is Spring Initializr?

Spring Initializr is a convenient way to create a Spring Boot project.We can go to the Spring Initializr site, choose a dependency management tool (either Maven or Gradle), a language (Java, Kotlin or Groovy), a packaging scheme (Jar or War), version and dependencies, and download the project.

This **creates a skeleton project for us** and saves setup time so that we can concentrate on adding business logic.

Even when we use our IDE's (such as STS or Eclipse with STS plugin) new project wizard to create a Spring Boot project, it uses Spring Initializr under the hood.

What Spring Boot Starters Are Available out There?

At the time of this writing, there are more than 50 starters at our disposal. The most commonly used are:

- *spring-boot-starter:* core starter, including auto-configuration support, logging, and YAML
- *spring-boot-starter-data-jpa:* starter for using Spring Data JPA with Hibernate
- *spring-boot-starter-security:* starter for using Spring Security
- *spring-boot-starter-test:* starter for testing Spring Boot applications

- *spring-boot-starter-web:* starter for building web, including RESTful, applications using Spring MVC

How to Deploy Spring Boot Web Applications as Jar and War Files?

Traditionally, we package a web application as a WAR file, then deploy it into an external server. Doing this allows us to arrange multiple applications on the same server. During the time that CPU and memory were scarce, this was a great way to save resources.

However, things have changed. Computer hardware is fairly cheap now, and the attention has turned to server configuration. A small mistake in configuring the server during deployment may lead to catastrophic consequences.

Spring tackles this problem by providing a plugin, namely *spring-boot-maven-plugin*, to package a web application as an executable JAR. To include this plugin, just add a *plugin* element to *pom.xml*:

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

With this plugin in place, we'll get a fat JAR after executing the *package* phase. This JAR contains all the necessary dependencies, including an embedded server. Thus, we no longer need to worry about configuring an external server.

We can then run the application just like we would an ordinary executable JAR.

Notice that the *packaging* element in the *pom.xml* file must be set to *jar* to build a JAR file:

```
<packaging>jar</packaging>
```

If we don't include this element, it also defaults to *jar*.

What Are Possible Sources of External Configuration?

Spring Boot provides support for external configuration, allowing us to run the same application in various environments. **We can use properties files, YAML files, environment variables, system properties, and command-line option arguments to specify configuration properties.**

We can then gain access to those properties using the *@Value* annotation, a bound object via the *@ConfigurationProperties* annotation, or the *Environment* abstraction.

What is Spring Boot Devtools Used For?

Spring Boot Developer Tools, or DevTools, is a set of tools making the development process easier. To include these development-time features, we just need to add a dependency to the *pom.xml* file:

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

The *spring-boot-devtools* module is automatically disabled if the application runs in production. The repackaging of archives also excludes this module by default. Hence, it won't bring any overhead to our final product.

By default, DevTools applies properties suitable to a development environment. These properties disable template caching, enable debug logging for the web group, and so on. As a result, we have this sensible development-time configuration without setting any properties.

Applications using DevTools restart whenever a file on the classpath changes. This is a very helpful feature in development, as it gives quick feedback for modifications.

By default, static resources, including view templates, don't set off a restart. Instead, a resource change triggers a browser refresh. Notice this can only happen if the LiveReload extension is installed in the browser to interact with the embedded LiveReload server that DevTools contains

What Is Spring Boot Actuator Used For?

Essentially, Actuator brings Spring Boot applications to life by enabling production-ready features. **These features allow us to monitor and manage applications when they're running in production**.

Integrating Spring Boot Actuator into a project is very simple. All we need to do is to include the *spring-boot-starter-actuator* starter in the *pom.xml* file:

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

Spring Boot Actuator can expose operational information using either HTTP or JMX endpoints. Most applications go for HTTP, though, where the identity of an endpoint and the */actuator* prefix form a URL path.

Here are some of the most common built-in endpoints Actuator provides:

- *env:* Exposes environment properties
- *health:* Shows application health information
- *httptrace:* Displays HTTP trace information
- *info:* Displays arbitrary application information
- *metrics:* Shows metrics information
- *loggers:* Shows and modifies the configuration of loggers in the application
- *mappings:* Displays a list of all *@RequestMapping* paths

Which Is a Better Way to Configure a Spring Boot Project – Using Properties or YAML?

YAML offers many advantages over properties files, such as:

- More clarity and better readability
- Perfect for hierarchical configuration data, which is also represented in a better, more readable format
- Support for maps, lists, and scalar types
- Can include several **profiles** in the same file (since Spring Boot 2.4.0, this is possible for properties files too)

However, writing it can be a little difficult and error-prone due to its indentation rules.

What Are the Basic Annotations that Spring Boot Offers?

The primary annotations that Spring Boot offers reside in its *org.springframework.boot.autoconfigure* and its sub-packages. Here are a couple of basic ones:

- *@EnableAutoConfiguration* – to make Spring Boot look for auto-configuration beans on its classpath and automatically apply them.
- *@SpringBootApplication* – used to denote the main class of a Boot Application. This annotation combines *@Configuration*, *@EnableAutoConfiguration*, and *@ComponentScan* annotations with their default attributes.

How Can You Change the Default Port in Spring Boot?
We can change the default port of a server embedded in Spring Boot using one of these ways:

- using a properties file – we can define this in an *application.properties* (or *application.yml*) file using the property *server.port*
- programmatically – in our main *@SpringBootApplication* class, we can set the *server.port* on the *SpringApplication* instance
- using the command line – when running the application as a jar file, we can set the server.port as a java command argument:
  `java -jar -Dserver.port=8081 myspringproject.jar`

Why Do We Need Spring Profiles?

When developing applications for the enterprise, we typically deal with multiple environments such as Dev, QA, and Prod. The configuration properties for these environments are different.

For example, we might be using an embedded H2 database for Dev, but Prod could have the proprietary Oracle or DB2. Even if the DBMS is the same across environments, the URLs would definitely be different.

To make this easy and clean, **Spring has the provision of profiles, to help separate the configuration for each environment**. So that instead of maintaining this programmatically, the properties can be kept in separate files such as *application-dev.properties* and *application-prod.properties*. The default *application.properties* points to the currently active profile using *spring.profiles.active* so that the correct configuration is picked up

What is a profile ?

Profiles are a core feature of the framework — **allowing us to map our beans to different profiles** — for example, *dev*, *test*, and *prod*.

We can then activate different profiles in different environments to bootstrap only the beans we need.

Use *@Profile* on a Bean

Let's start simple and look at how we can make a bean belong to a particular profile. **We use the *@Profile* annotation — we are mapping the bean to that**

particular profile; the annotation simply takes the names of one (or multiple) profiles.

Consider a basic scenario: We have a bean that should only be active during development but not deployed in production.

We annotate that bean with a *dev* profile, and it will only be present in the container during development. In production, the *dev* simply won't be active:

```
@Component
@Profile("dev")
public class DevDatasourceConfig
```

As a quick sidenote, profile names can also be prefixed with a NOT operator, e.g., *!dev*, to exclude them from a profile.

In the example, the component is activated only if *dev* profile is not active:

```
@Component
@Profile("!dev")
public class DevDatasourceConfig
```

Declare Profiles in XML

Profiles can also be configured in XML. The <*beans*> tag has a *profiles* attribute, which takes comma-separated values of the applicable profiles:

```
<beans profile="dev">
   <bean id="devDatasourceConfig"
     class="org.baeldung.profiles.DevDatasourceConfig" />
</beans>
```