Spring Boot Interview Questions

Last modified: February 18, 2021

by Nguyen Nam Thai

Spring Boot

Interview

1. Introduction

Since its introduction, Spring Boot has been a key player in the Spring ecosystem. This project makes our life much easier with its auto-configuration ability.

In this tutorial, we'll cover some of the most common questions related to Spring Boot that may come up during a job interview.

Further reading:

Top Spring Framework Interview Questions

A quick discussion of common questions about the Spring Framework that might come up during a job interview.

Read more →

A Comparison Between Spring and Spring Boot

Understand the difference between Spring and Spring Boot.

Read more →

2. Questions

Q1. 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

Q2. What Are the Differences Between Spring and Spring Boot?

The Spring Framework provides multiple features that make the development of web applications easier. These features include dependency injection, data binding, aspect-oriented programming, data access, and many more.

Over the years, Spring has been growing more and more complex, and the amount of configuration such application requires can be intimidating. This is where Spring Boot comes in handy – it makes configuring a Spring application a breeze.

Essentially, while Spring is unopinionated, **Spring Boot takes an opinionated view of the platform and libraries, letting us get started quickly.**

Here are two of the most important benefits Spring Boot brings in:

- Auto-configure applications based on the artifacts it finds on the classpath
- Provide non-functional features common to applications in production, such as security or health checks

Please check one of our other tutorials for a detailed comparison between vanilla Spring and Spring Boot.

Q3. How Can We Set up a Spring Boot Application With Maven?

We can include Spring Boot in a Maven project just like we would any other library. However, the best way is to inherit from the *spring-boot-starter-parent* project and declare dependencies to Spring Boot starters. Doing this lets our project reuse the default settings of Spring Boot.

Inheriting the *spring-boot-starter-parent* project is straightforward – we only need to specify a *parent* element in *pom.xml*:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0.RELEASE</version>
</parent>
```

We can find the latest version of *spring-boot-starter-parent* on Maven Central.

Using the starter parent project is convenient, but not always feasible. For instance, if our company requires all projects to inherit from a standard POM, we can still benefit from Spring Boot's dependency management using a custom parent.

Q4. 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.

Q5. What Spring Boot Starters Are Available out There?

Each starter plays a role as a one-stop-shop for all the Spring technologies we need. Other required dependencies are then transitively pulled in and managed in a consistent way.

All starters are under the *org.springframework.boot* group and their names start with *spring-boot-starter-*. **This naming pattern makes it easy to find starters, especially when working with IDEs that support searching dependencies by name.**

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-aop:* starter for aspect-oriented programming with Spring AOP and AspectJ
- *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

For a complete list of starters, please see this repository.

To find more information about Spring Boot starters, take a look at Intro to Spring Boot Starters.

Q6. How to Disable a Specific Auto-Configuration?

If we want to disable a specific auto-configuration, we can indicate it using the *exclude* attribute of the *@EnableAutoConfiguration* annotation. For instance, this code snippet neutralizes *DataSourceAutoConfiguration*:

```
// other annotations
@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)
public class MyConfiguration { }
```

If we enabled auto-configuration with the *@SpringBootApplication* annotation — which has *@EnableAutoConfiguration* as a meta-annotation — we could disable auto-configuration with an attribute of the same name:

```
// other annotations
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
public class MyConfiguration { }
```

We can also disable an auto-configuration with the *spring.autoconfigure.exclude* environment property. This setting in the *application.properties* file does the same thing as before:

```
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConf
iguration
```

Q7. How to Register a Custom Auto-Configuration?

To register an auto-configuration class, we must have its fully-qualified name listed under the *EnableAutoConfiguration* key in the *META-INF/spring.factories* file:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.baeldung.autoconfigure.C
ustomAutoConfiguration
```

If we build a project with Maven, that file should be placed in the *resources/META-INF* directory, which will end up in the mentioned location during the *package* phase.

Q8. How to Tell an Auto-Configuration to Back Away When a Bean Exists?

To instruct an auto-configuration class to back off when a bean is already existent, we can use the *@ConditionalOnMissingBean* annotation. The most noticeable attributes of this annotation are:

- *value:* The types of beans to be checked
- *name:* The names of beans to be checked

When placed on a method adorned with *@Bean*, the target type defaults to the method's return type:

```
@Configuration
public class CustomConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public CustomService service() { ... }
}
```

Q9. 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*.

In case we want to build a WAR file, change the *packaging* element to *war*:

```
<packaging>war</packaging>
```

And leave the container dependency off the packaged file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
```

After executing the Maven *package* phase, we'll have a deployable WAR file.

Q10. How to Use Spring Boot for Command Line Applications?

Just like any other Java program, a Spring Boot command line application must have a *main* method. This method serves as an entry point, which invokes the *SpringApplication#run* method to bootstrap the application:

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class);
        // other statements
    }
}
```

The *SpringApplication* class then fires up a Spring container and auto-configures beans.

Notice we must pass a configuration class to the *run* method to work as the primary configuration source. By convention, this argument is the entry class itself.

After calling the *run* method, we can execute other statements as in a regular program.

Q11. 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.

Q12. What Does it Mean that Spring Boot Supports Relaxed Binding?

Relaxed binding in Spring Boot is applicable to the type-safe binding of configuration properties.

With relaxed binding, **the key of a property doesn't need to be an exact match of a property name.** Such an environment property can be written in camelCase, kebab-case, snake_case, or in uppercase with words separated by underscores.

For example, if a property in a bean class with the *@ConfigurationProperties* annotation is named *myProp*, it can be bound to any of these environment properties: *myProp*, *my-prop*, *my_prop*, or *MY_PROP*.

Q13. 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.

For further information on this topic, please see Overview of Spring Boot DevTools.

Q14. How to Write Integration Tests?

When running integration tests for a Spring application, we must have an *ApplicationContext*.

To make our life easier, Spring Boot provides a special annotation for testing – *@SpringBootTest*. This annotation creates an *ApplicationContext* from configuration classes indicated by its *classes* attribute.

In case the *classes* attribute isn't set, Spring Boot searches for the primary configuration class. The search starts from the package containing the test up until it finds a class annotated with *@SpringBootApplication* or *@SpringBootConfiguration*.

For detailed instructions, check out our tutorial on testing in Spring Boot.

Q15. 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

Please refer to our Spring Boot Actuator tutorial for a detailed rundown.

Q16. 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.

For details and working samples, please refer to our Spring YAML vs Properties tutorial.

Q17. 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.

Spring Boot Annotations offers more insight into the subject.

Q18. 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
```

Q19. Which Embedded Servers does Spring Boot Support, and How to Change the Default?

As of date, **Spring MVC supports Tomcat, Jetty, and Undertow**. Tomcat is the default application server supported by Spring Boot's *web* starter.

Spring WebFlux supports Reactor Netty, Tomcat, Jetty, and Undertow with Reactor Netty as default.

In Spring MVC, to change the default, let's say to Jetty, we need to exclude Tomcat and include Jetty in the dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Similarly, to change the default in WebFlux to UnderTow, we need to exclude Reactor Netty and include UnderTow in the dependencies.

"Comparing embedded servlet contains in Spring Boot" contains more details on the different embedded servers we can use with Spring MVC.

Q20. 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.

Spring Profiles gives a comprehensive view of this topic.

3. Conclusion

This tutorial went over some of the most critical questions on Spring Boot that you may face during a technical interview. We hope they will help you land your dream job.

Comments are closed on this article!