Spring Boot Interview Questions

Last modified: January 11, 2019

by Nguyen Nam Thai (/author/namthai-nguyen/)

Spring Boot (/category/spring/spring-boot/)

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.

2. Questions

Q1. 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 (https://www.baeldung.com/spring-vs-spring-boot).

Q2. 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 (https://www.baeldung.com/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*:

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.1.RELEASE</version>
</parent>
```

We can find the latest version of *spring-boot-starter-parent* on Maven Central (https://search.maven.org/search?q=g:org.springframework.boot%20AND%20a:spring-boot-starter-parent&core=gav).

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 cannot rely on the Spring Boot starter parent.

In this case, we can still get the benefits of dependency management with this POM element:

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.1.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Finally, we can add some dependencies to Spring Boot starters, and then we're good to go.

Q3. What Spring Boot starters are available out there?

Dependency management is a crucial facet of any project. When a project is complex enough, managing dependencies may turn into a nightmare, as there will be too many artifacts involved.

This is where Spring Boot starters come in handy. 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-jdbc:* starter for using JDBC with the HikariCP connection pool
- *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 (https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-starters).

To find more information about Spring Boot starters, take a look at Intro to Spring Boot Starters (https://www.baeldung.com/spring-boot-starters).

Q4. 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
```

Q5. 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.baeld
```

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.

Q6. 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:

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

Q7. 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*:

```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:

```xml
<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.

Q8. 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.

Q9. 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 (https://www.baeldung.com/configuration-properties-in-spring-boot), or the *Environment* abstraction.

Here are the most common sources of external configuration:

- Command-line properties: Command-line option arguments are program arguments starting with a double hyphen, such as *–server.port=8080*. Spring Boot converts all the arguments to properties and adds them to the set of environment properties.
- Application properties: Application properties are those loaded from the *application.properties* file or its YAML counterpart. By default, Spring Boot searches for this file in the current directory, classpath root, or their *config* subdirectory.
- Profile-specific properties: Profile-specific properties are loaded from the *application-{profile}.properties* file or its YAML counterpart. The *{profile}* placeholder refers to an active profile. These files are in the same locations as, and take precedence over, non-specific property files.

Q10. 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 (https://www.baeldung.com/configuration-properties-in-spring-boot).

With relaxed binding, the key of an environment 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*.

Q11. 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:

```xml
<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 (https://www.baeldung.com/spring-boot-devtools).

Q12. 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*.

Notice if we use JUnit 4, we must decorate the test class with *@RunWith(SpringRunner.class)*.

For detailed instructions, check out our tutorial on testing in Spring Boot (https://www.baeldung.com/spring-boot-testing).

Q13. 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:

```xml
<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:

- *auditevents:* Exposes audit events information
- *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
- *scheduledtasks:* Displays the scheduled tasks in your application
- *threaddump:* Performs a thread dump

Please refer to our Spring Boot Actuator tutorial (https://www.baeldung.com/spring-boot-actuators) for a detailed rundown.

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.

CATEGORIES

SERIES

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)