List of Spring Boot Interview Questions and Answers

Q1. What is the Spring Boot?
Q2. Explain a few important Spring Boot Key features?
Q3. What is Spring Boot Auto-configuration?
Q4. How Spring boot internally works (Explain the run() method in Spring boot)?
Q5. What are the different ways to create a Spring boot application?
Q6. Explain @SpringBootApplication,@Configuration, and @ComponentScan annotations
Q7. What are Spring boot starters and name a few important Spring boot starter dependencies?
Q8. How does Spring Enable Creating Production-Ready Applications in Quick Time?
Q9. What Is the Minimum Baseline Java Version for Spring Boot 3?
Q10. What are Different Ways of Running Spring Boot Application?
Q11. Name all Spring Boot Annotations?
Q12. What Is the Difference Between @SpringBootApplication and @EnableAutoConfiguration Annotation?
Q13. Why do we need a spring-boot-maven plugin?
Q14. What is the Spring Boot Actuator and its Features?
Q15. How to Use Jetty Instead of Tomcat in Spring-Boot-Starter-Web?
Q16. How to generate a WAR file with Spring Boot?
Q17. How many types of projects we can create using Spring boot?
Q18. How to Change Default Embedded Tomcat Server Port and Context Path in Spring Boot Application?
Q19. What Embedded servers does Spring Boot support?
Q20. How to use logging with Spring Boot?
Q21. What is the Spring Boot Starter Parent and How to Use it?
Q22. How to Write Integration Tests in Spring Boot Application?
Q23. How to Test Spring MVC Controllers?
Q24. @SpringBootTest vs @WebMvcTest?
Q25. How to Implement Security for Spring Boot Application?

Q1. What is the Spring Boot?

Spring Boot is basically an extension of the Spring framework which eliminated the boilerplate configurations required for setting up a Spring application.

Spring Boot is an opinionated framework that helps developers build Spring-based applications quickly and easily. **The main goal of Spring Boot is to quickly create**

Spring-based applications without requiring developers to write the same boilerplate configuration again and again.

Q2. Explain a few important Spring Boot Key features?

Let me a list of a few key features of the Spring boot and we will discuss each key feature briefly.

1. Spring Boot starters
2. Spring Boot autoconfiguration
3. Elegant configuration management
4. Spring Boot Actuator
5. Easy-to-use embedded servlet container support

1. Spring Boot Starters

Spring Boot offers many starter modules to get started quickly with many of the commonly used technologies, like SpringMVC, JPA, MongoDB, Spring Batch, SpringSecurity, Solr, ElasticSearch, etc. These starters are pre-configured with the most commonly used library dependencies so you don't have to search for compatible library versions and configure them manually.

For example, the `spring-boot-starter-data-jpa` starter module includes all the dependencies required to use Spring Data JPA, along with Hibernate library dependencies, as Hibernate is the most commonly used JPA implementation.

One more example, when we add the `spring-boot-starter-web` dependency, it will by default pull all the commonly used libraries while developing Spring MVC applications, such as `spring-webmvc`, `jackson-json`, `validation-api`, and `tomcat`.

Not only does the spring-boot-starter-web add all these libraries but it also configures the commonly registered beans like `DispatcherServlet`, `ResourceHandlers`, `MessageSource`, etc. with sensible defaults.

2. Spring Boot Autoconfiguration

Spring Boot addresses the problem that Spring applications need complex configuration by eliminating the need to manually set up the boilerplate configuration.

Spring Boot takes an opinionated view of the application and configures various components automatically, by registering beans based on various criteria. The criteria can be:

- Availability of a particular class in a classpath
- Presence or absence of a Spring bean
- Presence of a system property
- An absence of a configuration file

For example, if you have the `spring-webmvc` dependency in your classpath, Spring Boot assumes you are trying to build a SpringMVC-based web application and automatically tries to register `DispatcherServlet` if it is not already registered.

If you have any embedded database drivers in the classpath, such as H2 or HSQL, and if you haven't configured a `DataSource` bean explicitly, then Spring Boot will automatically register a `DataSource` bean using in-memory database settings.

3. Elegant Configuration Management

Spring Boot allows you to externalize configuration properties, such as database connection details, server port, and logging settings. It supports various configuration formats like properties files, YAML files, environment variables, and more. The externalized configuration makes it easier to configure and manage application properties in different environments.

4. Spring Boot Actuator

Being able to get the various details of an application running in production is crucial to many applications. The Spring Boot actuator provides a wide variety of such production-ready features without requiring developers to write much code. Some of the Spring actuator features are:

- Can view the application bean configuration details
- Can view the application URL mappings, environment details, and configuration parameter values
- Can view the registered health check metrics

5. Easy-to-Use Embedded Servlet Container Support

Traditionally, while building web applications, you need to create `WAR` type modules and then deploy them on external servers like `Tomcat`, `WildFly`, etc. But by using Spring Boot, you can create a `JAR` type module and embed the servlet container in the application very easily so that the application will be a self-contained deployment unit.

Also, during development, you can easily run the Spring Boot JAR type module as a Java application from the IDE or from the command line using a build tool like **Maven** or Gradle.

Q3. What is Spring Boot Auto-configuration?

Spring Boot auto-configuration attempts to automatically configure your Spring application
based on the jar dependencies that you have added.

Why do we need Spring Boot Auto Configuration?
-> Spring-based applications have a lot of configuration.
-> When we use Spring MVC, we need to configure
    - Component scan,
    - Dispatcher Servlet
    - View resolver
    - Web jars(for delivering static content) among other things.
-> When we use Hibernate/JPA, we would need to configure a
    - data source
    - entity manager factory/session factory
    - transaction manager among a host of other things.
-> When you use cache
    - Cache configuration
-> When you use Message Queue
    - Message queue configuration
-> When you use a NoSQL database
    - NoSQL database configuration

Spring Boot: Can we think differently?
Spring Boot brings in a new thought process around this:
-> Can we bring more intelligence into this? When a spring MVC jar is added to an application,
  can we auto-configure some beans automatically?
-> How about auto-configuring a Data Source if Hibernate jar is on the classpath?
-> How about auto-configuring a Dispatcher Servlet if the Spring MVC jar is on the classpath?

One more example, if HSQLDB is present on your classpath and you have not configured any database manually, Spring will auto-configure an in-memory database for you.

The Spring Boot auto-configuration feature tries to automatically configure your Spring application based upon the JAR dependency you have added in the classpath.

Q4. How Spring boot internally works (Explain the run() method in Spring boot)?

The below 10 steps show the internal working of the run() method:

1. Spring boot application execution will start from the main() method
2. The main() method internally call SpringApplication.run() method
3. SpringApplication.run() method performs bootstrapping for our spring boot application
4. Starts StopWatch to identify the time taken to bootstrap the spring boot application
5. Prepares environment to run our spring boot application (dev, prod, qa, uat)
6. Print banner (Spring Boot Logo prints on console)
7. Start the IOC container (ApplicationContext) based on the classpath (default, Web servlet/ Reactive)
8. Refresh context
9. Trigger Runners (ApplicationRunner or CommandLineRunner)
10. Return ApplicationContext reference (Spring IOC)

Q5. What are the different ways to create a Spring boot application?

Different ways to create Spring boot project:
1. Using Spring Initializr - Create a Spring boot project using Spring Initializr and import in any IDE - Eclipse STS, Eclipse, IntelliJ idea, VSCode, Netbeans

2. Using Spring Starter Project in STS (Eclipse) - You can directly create a Spring boot project in STS using the Spring Starter Project option.

3. Spring Boot CLI - The Spring Boot CLI is a command-line tool that you can use if you want to quickly develop a Spring application.

Q6. Explain @SpringBootApplication, @Configuration and @ComponentScan annotations

The **@SpringBootApplication** annotation indicates a configuration class that declares one or more **@Bean** methods and also triggers auto-configuration and component scanning. This is a convenience annotation that is equivalent to declaring below three annotations:

@SpringBootApplication = **@Configuration + @EnableAutoConfiguration + @ComponentScan**

@Configuration: The `@Configuration` annotation indicates that the class contains bean configurations and should be processed by the Spring IoC container. It allows you to define beans and their dependencies using annotations like `@Bean`.

@EnableAutoConfiguration: The `@EnableAutoConfiguration` annotation enables Spring Boot's auto-configuration mechanism. It automatically configures the Spring application based on the classpath dependencies, project settings, and environment. Auto-configuration eliminates the need for manual configuration and reduces boilerplate code.

@ComponentScan: The `@ComponentScan` annotation tells Spring where to look for components (such as controllers, services, and repositories) to be managed by the Spring IoC container. It scans the specified packages and registers the annotated classes as beans.

Q7. What are Spring boot starters and name a few important Spring boot starter dependencies?

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 technology that you need, without having to hunt through sample code and copy-paste loads of dependency descriptors.

For example, while developing the REST service or web application; we can use libraries like Spring MVC, Tomcat, and Jackson – a lot of dependencies for a single application. the **spring-boot-starter-web** starter can help to reduce the number of manually added dependencies just by adding a **spring-boot-starter-web** dependency.

So instead of manually specifying the dependencies just add one **spring-boot-starter-web** starter as in the following example:

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

Few commonly used Spring boot starters:

`spring-boot-starter`: core starter, including auto-configuration support, logging, and YAML

`spring-boot-starter-aop`: for aspect-oriented programming with Spring AOP and AspectJ

`spring-boot-starter-data-jpa`: for using Spring Data JPA with Hibernate

`spring-boot-starter-security`: for using Spring Security

`spring-boot-starter-test`: for testing Spring Boot applications

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

`spring-boot-starter-data-mongodb`: Starter for using MongoDB document-oriented database and Spring Data MongoDB

`spring-boot-starter-data-rest`: Starter for exposing Spring Data repositories over REST using Spring Data REST

`spring-boot-starter-webflux`: Starter for building WebFlux applications using Spring Framework's Reactive Web support

Q8. How does Spring Enable Creating Production-Ready Applications in a Quick Time?

Spring Boot aims to enable production-ready applications in a quick time. Spring Boot provides a few non-functional features out of the box like caching, logging, monitoring, and embedded servers.

1. **spring-boot-starter-actuator** - To use advanced features like monitoring & tracing to your application out of the box
2. **spring-boot-starter-undertow, spring-boot-starter-jetty, spring-boot-starter-tomcat** - To pick your specific choice of Embedded Servlet Container
3. **spring-boot-starter-logging** - For Logging using log back
4. **spring-boot-starter-cache** - Enabling Spring Framework's caching support

Q9. What Is the Minimum Baseline Java Version for Spring Boot 3?

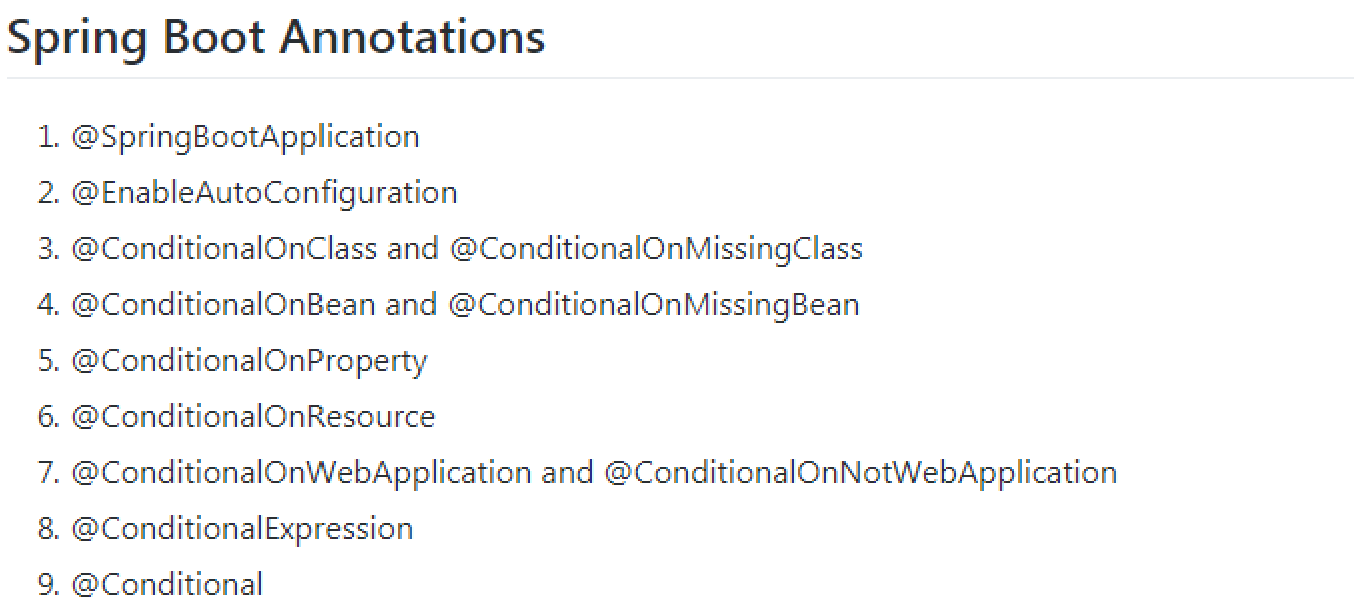
Spring Boot 3.0 requires Java 17 or later. It also requires Spring Framework 6.0.

Q10. What are Different Ways of Running Spring Boot Application?

Spring Boot offers several ways of running Spring Boot applications. I would like to suggest five ways we can run the Spring Boot Application

1. Running from an IDE
2. Running as a Packaged Application
3. Using the Maven Plugin
4. Using External Tomcat
5. Using the Gradle Plugin

Q11. Name all Spring Boot Annotations?



Spring Boot Annotations

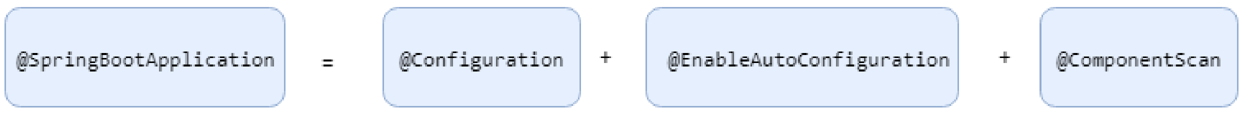
1. @SpringBootApplication
2. @EnableAutoConfiguration
3. @ConditionalOnClass and @ConditionalOnMissingClass
4. @ConditionalOnBean and @ConditionalOnMissingBean
5. @ConditionalOnProperty
6. @ConditionalOnResource
7. @ConditionalOnWebApplication and @ConditionalOnNotWebApplication
8. @ConditionalExpression
9. @Conditional

Q12. What Is the Difference Between @SpringBootApplication and @EnableAutoConfiguration Annotation?

@EnableAutoConfiguration is to enable the automatic configuration feature of the Spring Boot application which automatically configures things if certain classes are present in Classpath. For example, it can configure *`Thymeleaf`*, *`TemplateResolver`*, and *`ViewResolver`* if **Thymeleaf** is present in the classpath.

@EnableAutoConfiguration also combines **@Configuration** and **@ComponentScan** annotations to enable Java-based configuration and component scanning in your project

On the other hand, @SpringBootApplication annotation indicates a configuration class that declares one or more **@Bean** methods and also triggers auto-configuration and component scanning. This is a convenience annotation that is equivalent to declaring **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan**.



Q13. Why do we need a spring-boot-maven plugin?

The Spring Boot Maven plugin provides many convenient features:

- It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.
- It searches for the public static void `main()` method to flag as a runnable class.
- It provides a built-in dependency resolver that sets the version number to match Spring Boot dependencies. You can override any version you wish, but it will default to Boot’s chosen set of versions.

The Spring Boot Plugin has the following goals.

- `spring-boot:run` runs your Spring Boot application.
- `spring-boot:repackage` repackages your jar/war to be executable.
- `spring-boot:start` and `spring-boot:stop` to manage the lifecycle of your Spring Boot application (i.e. for integration tests).
- `spring-boot:build-info` generates build information that can be used by the Actuator.

Q14. What is the Spring Boot Actuator and its Features?

Spring Boot Actuator provides production-ready features for monitoring and managing Spring Boot applications. It offers a set of built-in endpoints and metrics that allow you to gather valuable insights into the health, performance, and management of your application.

Here are some key features provided by Spring Boot Actuator:
Health Monitoring: The actuator exposes a `/health` endpoint that provides information about the health status of your application. It can indicate whether your application is up and running, any potential issues, and detailed health checks for different components, such as the database, cache, and message brokers.

Metrics Collection: The actuator collects various metrics about your application's performance and resource utilization. It exposes endpoints like `/metrics` and `/prometheu`s to retrieve information about HTTP request counts, memory usage, thread pool statistics, database connection pool usage, and more. These metrics can be integrated with monitoring systems like Prometheus, Graphite, or Micrometer.

Auditing and Tracing: Actuator allows you to track and monitor the activities happening within your application. It provides an *`/auditevents`* endpoint to view audit events like login attempts, database changes, or any custom events. Additionally, Actuator integrates with distributed tracing systems like Zipkin or Spring Cloud Sleuth to trace requests as they flow through different components.

Environment Information: The actuator exposes an *`/info`* endpoint that displays general information about your application, such as version numbers, build details and any custom information you want to include. It is useful for providing diagnostic details about your application in runtime environments.

Configuration Management: Actuator provides an *`/configprops`* endpoint that lists all the configuration properties used in your application. It helps in understanding the current configuration state and identifying potential issues or inconsistencies.

Remote Management: Actuator allows you to manage and interact with your application remotely. It provides various endpoints, such as *`/shutdown`* to gracefully shut down the application, *`/restart`* to restart the application, and *`/actuator`* to list all available endpoints. These endpoints can be secured using Spring Security for proper access control.

Custom Endpoints: Actuator allows you to define custom endpoints to expose additional management or monitoring information specific to your application. You can create your own custom Actuator endpoints by extending the AbstractEndpoint class or implementing the Endpoint interface.

Spring Boot Actuator is a powerful tool that facilitates the monitoring, management, and troubleshooting of Spring Boot applications. Its features make it easier to gain insights into the health, performance, and behavior of your application, enabling efficient monitoring and effective management in production environments.

Enabling the Actuator

The simplest way to enable the features is to add a dependency to the **spring-boot-starter-actuator** ‘Starter’. To add the actuator to a Maven-based project, add the following ‘Starter’ dependency:

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

For Gradle, use the following declaration:

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-actuator")
}
```

Q15. How to Use Jetty Instead of Tomcat in Spring-Boot-Starter-Web?

Remove the existing default tomcat dependency from *spring-boot-starter-web* and add the *spring-boot-starter-jetty* dependency:

```
<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>
```

Q16. How to generate a WAR file with Spring Boot?

I suggest below three steps to generate and deploy the Spring Boot WAR file.

1. Change the packaging type.

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

2. Add **spring-boot-starter-tomcat** as the **provided** scope

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

3. Spring Boot Application or *`Main`* class extends *`SpringBootServletInitializer`*

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication
public class Springboot2WebappJspApplication extends SpringBootServletInitializer{

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        return application.sources(Springboot2WebappJspApplication.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(Springboot2WebappJspApplication.class, args);
    }
}
```

Q17. How many types of projects we can create using Spring Boot?

We can create 3 types of projects using Spring boot starter dependencies.

3 types of Spring boot applications:

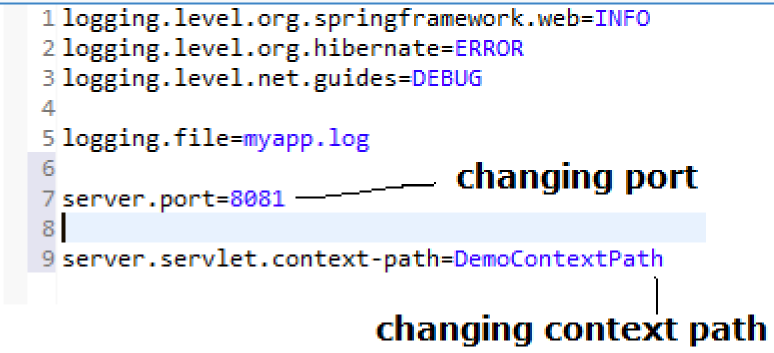
1. If *`we have a spring-boot-starter`* dependency in a classpath then the spring boot application comes under the **default** category.
2. If we have *`spring-boot-starter-web`* dependency in a classpath then the spring boot application comes under the **servlet** category.
3. If we have a *`spring-boot-starter-webflux`* dependency in a classpath then the spring boot application comes under the **reactive** category.

Q18. How to Change Default Embedded Tomcat Server Port and Context Path in Spring Boot Application?

By default, the embedded tomcat server starts on port 8080 and by default, the context path is "/". Now let's change the default port and context path by defining properties in an **application.properties** file -

/src/main/resources/application.properties

```
server.port=8080
server.servlet.context-path=/springboot2webapp
```



```
logging.level.org.springframework.web=INFO
logging.level.org.hibernate=ERROR
logging.level.net.guides=DEBUG

logging.file=myapp.log

server.port=8081

server.servlet.context-path=DemoContextPath
```

Q19. What Embedded servers does Spring Boot support?

Spring Boot provides support for several embedded servers out-of-the-box. These embedded servers allow you to package your Spring Boot application as a standalone executable JAR file, containing the application and the server runtime.

Here are the embedded servers supported by Spring Boot:
Apache Tomcat: Tomcat is the default embedded server in Spring Boot. It provides a robust and widely used HTTP server and servlet container. Spring Boot uses Tomcat as the default embedded server when you include the spring-boot-starter-web dependency.

Jetty: Jetty is another popular choice for embedded servers. It is lightweight, fast, and has a small memory footprint. Spring Boot provides support for Jetty through the spring-boot-starter-jetty dependency.

Undertow: Undertow is a high-performance web server designed for modern applications. It is known for its scalability and low resource consumption. Spring Boot offers support for Undertow through the spring-boot-starter-undertow dependency.

By default, when you create a Spring Boot application, it uses Apache Tomcat as the embedded server. However, you can easily switch to Jetty or Undertow by excluding the Tomcat dependency and including the desired server dependency in your project's build configuration.

Q20. How to use logging with Spring Boot?

We can use logging with Spring Boot by specifying log levels on the **application.properties** file. Spring Boot loads this file when it exists in the classpath and it can be used to configure both Spring Boot and application code.

Spring Boot, by default, includes **spring-boot-starter-logging** as a transitive dependency for the **spring-boot-starter** module. By default, Spring Boot includes *`SLF4J`* along with *`Logback`* implementations.

If *`Logback`* is available, Spring Boot will choose it as the logging handler. You can easily configure logging levels within the application.properties file without having to create logging provider-specific configuration files such as *`logback.xml`* or *`log4j.properties`*.

```
logging.level.org.springframework.web=INFO
logging.level.org.hibernate=ERROR
logging.level.net.guides=DEBUG
```

Q21. What is the Spring Boot Starter Parent and How to Use it?

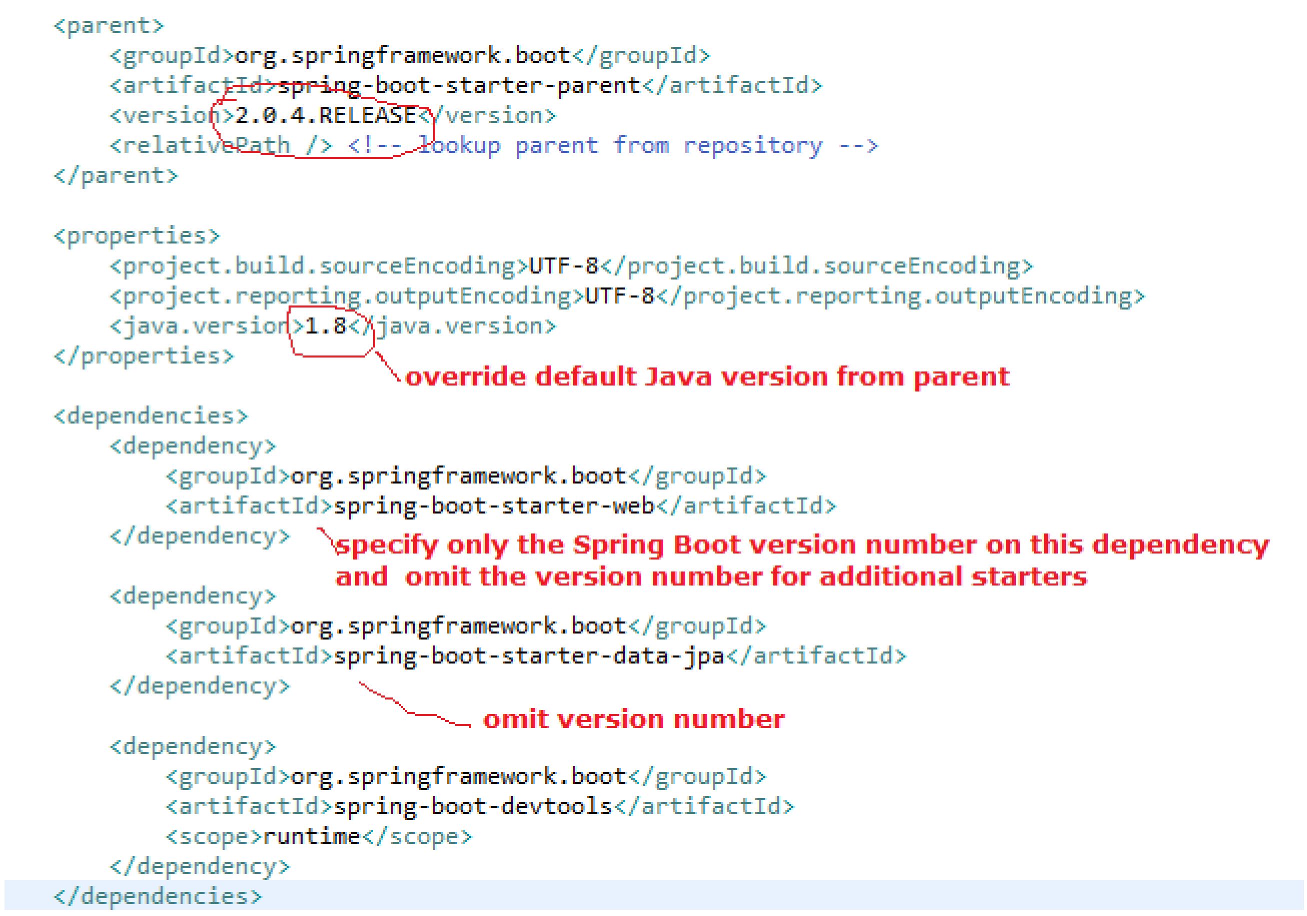
All Spring Boot projects typically use `spring-boot-starter-parent` as the parent in `pom.xml`.

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.4.RELEASE</version>
    </parent>
```

`spring-boot-starter-parent` allows us to manage the following things for multiple child projects and modules:

- Configuration - Java Version and Other Properties
- Dependency Management - Version of dependencies
- Default Plugin Configuration

We should need to specify only the Spring Boot version number on this dependency. If you import additional starters, you can safely omit the version number.



```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.4.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

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

override default Java version from parent
specify only the Spring Boot version number on this dependency and omit the version number for additional starters
omit version number

Q22. How to Write Integration Tests in Spring Boot Application?

As the name suggests, integration tests focus on integrating different layers of the application. That also means no mocking is involved.

Spring Boot provides `@SpringBootTest` annotation for Integration testing. This annotation creates an application context and loads the full application context.

`@SpringBootTest` will bootstrap the full application context, which means we can `@Autowire` any bean that's picked up by component scanning into our Integration tests.

Q23. How to Test Spring MVC Controllers?

SpringBoot provides `@WebMvcTest` annotation to test Spring MVC Controllers. Also, the @WebMvcTest annotation-based test runs faster because it will load only the specified controller and its dependencies only without loading the entire application.

Q24. @SpringBootTest vs @WebMvcTest?

`@SpringBootTest` annotation loads the full application context so that we can able to test various components. So basically, the `@SpringBootTest` annotation tells Spring Boot to look for the main configuration class (one with @SpringBootApplication, for instance) and use that to start a Spring application context.

`@WebMvcTest` annotation loads only the specified controller and its dependencies only without loading the entire application. For example, let's say you have multiple Spring MVC controllers in your Spring boot project - EmployeeController, UserController, LoginController, etc then we can use @WebMvcTest annotation to test only specific Spring MVC controllers without loading all the controllers and their dependencies.

Spring Boot provides `@SpringBootTest` annotation for Integration testing.

Spring boot provides `@WebMvcTest` annotation for testing Spring MVC controllers (Unit testing).

Q25. How to Implement Security for Spring Boot Application?

Spring boot provided auto-configuration of spring security for a quick start. Adding the Spring Security Starter (spring-boot-starter-security) to a Spring Boot application will:

- Enable HTTP basic security
- Register the `AuthenticationManager` bean with an in-memory store and a single user

- Ignore paths for commonly used static resource locations (such as /css/**, /js/**, /images/**, etc.)
- Enable common low-level features such as XSS, CSRF, caching, etc.

Add the below dependencies to the *pom.xml* file

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

Now if you run the application and access **http://localhost:8080**, you will be prompted to enter the user credentials. The default user is the `user` and the password is auto-generated. You can find it in the console log.

Using default security password: `78fa095d-3f4c-48b1-ad50-e24c31d5cf35`

You can change the default user credentials in *application.properties* as follows:

```
security.user.name=admin
security.user.password=secret
security.user.role=USER,ADMIN
```

Q26. What is the difference between @RestController and @Controller in Spring Boot?

In Spring Boot, both @RestController and @Controller are annotations used to define components that handle HTTP requests. However, there is a slight difference in their behavior and purpose :

@Controller: The @Controller annotation is used to mark a class as a Spring MVC controller. It is typically used in traditional Spring MVC applications. Controllers annotated with @Controller are responsible for handling requests and returning a response. These controllers typically return a view name or a ModelAndView object, which is resolved by a view resolver to render the appropriate view.

Example:

```java
@Controller
public class MyController {
    @GetMapping("/hello")
    public String sayHello() {
        return "hello";
    }
}
```

@RestController: The @RestController annotation is an extension of the @Controller annotation and is specifically tailored for RESTful web services. It combines the functionality of *@Controller* and *@ResponseBody*. Controllers annotated with @RestController return the response directly as the body of the HTTP response, typically in JSON or XML format.

Example:

```java
@RestController
public class MyRestController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, world!";
    }
}
```

The key difference is that @RestController eliminates the need for annotating individual request-handling methods with @ResponseBody. Every method in a @RestController is assumed to be a @ResponseBody by default, simplifying the process of building RESTful APIs.