**Spring Boot Interview Questions:**

Spring Boot is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

In short, Spring Boot is the combination of **Spring Framework** and **Embedded Servers**.

In Spring Boot, there is no requirement for XML configuration (deployment descriptor). It uses convention over configuration software design paradigm that means it decreases the effort of the developer.

We should use Spring Boot Framework because:

* The dependency injection approach is used in Spring Boot.
* It contains powerful database transaction management capabilities.
* It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc.
* It reduces the cost and development time of the application.

**1) Spring Boot Features? OR Advantage of using Spring Boot?**

* It creates **stand-alone** Spring applications that can be started using Java **-jar**.
* It tests web applications easily with the help of different **Embedded** HTTP servers such as **Tomcat, Jetty,** etc. We don't need to deploy WAR files.
* It provides opinionated '**starter**' POMs to simplify our Maven configuration.
* It provides **production-ready** features such as **metrics, health checks,** and **externalized configuration**.
* There is no requirement for **XML** configuration.
* It offers a **CLI** tool for developing and testing the Spring Boot application.
* It offers the number of **plug-ins**.
* It also minimizes writing multiple **boilerplate codes** (the code that has to be included in many places with little or no alteration), XML configuration, and annotations.
* It **increases productivity** and reduces development time.

## **Limitations of Spring Boot**

Spring Boot can use dependencies that are not going to be used in the application. These dependencies increase the size of the application.

The main goal of Spring Boot is to reduce **development, unit test,** and **integration test** time.

* Provides Opinionated Development approach
* Avoids defining more Annotation Configuration
* Avoids writing lots of import statements
* Avoids XML Configuration.

By providing or avoiding the above points, Spring Boot Framework reduces **Development time, Developer Effort,** and **increases productivity**.

**Q) Spring Boot DevTools provides the following features:**

* **Property Defaults**
* **Automatic Restart**
* **LiveReload**
* **Remote Debug Tunneling**
* **Remote Update and Restart**

**2) What is @SpringBootApplication?**

# **🡪**

@SpringBootApplication Annotation

**@SpringBootApplication**annotationis a convenience annotation introduced from [Spring Boot 1.2.0](https://javabeat.net/spring-boot/). If you have worked on the earlier spring boot versions, it is common that main class always annotate with the following [annotations](https://javabeat.net/annotations-in-java-5-0/):

* [**@Configuration**](https://javabeat.net/javaconfig-spring-3-0/)**:** This annotation is not specific to the spring boot applications. This annotation tags the class as the source for bean definitions. In short, this annotation is used for defining beans using the Java configuration.
* **@EnableAutoConfiguration :** This is a spring boot annotation. This annotation enables the application to add the beans using the classpath definitions.
* [**@ComponentScan**](https://javabeat.net/spring-mvc-component-scan-annotations/)**:** This annotation tells the spring to look for other components, configurations and services in the specified path.

The above three annotations have become quite common for all the spring boot application main class. This makes it worthwhile to [add @SpringBootApplication as a single annotation](https://github.com/spring-projects/spring-boot/issues/1842) that represents all the above three annotations.

**4) What are profiles in Spring boot?**

## **🡪 What Are Profiles?**

Every enterprise application has many environments, like:

Dev | Test | Stage | Prod | UAT / Pre-Prod

Each environment requires a setting that is specific to them. For example, in DEV, we do not need to constantly check database consistency. Whereas in TEST and STAGE, we need to. These environments host specific configurations called Profiles.

## **How Do we Maintain Profiles?**

This is simple — properties files! We make properties files for each environment and set the profile in the application accordingly, so it will pick the respective properties file.

We have used the @Profile("Dev")   to let the system know that this is the BEAN  that should be picked up when we set the application profile to DEV. The other two beans will not be created at all.

Application-dev.properties 🡺 spring.profiles.active=dev

**5) What is ResponseEntity class?**

**🡪** ResponseEntity Class ResponseEntity Class is**used to represent whole HTTP responses**. It supports the the message body, headers, status, etc  
You have more control over response by including body, headers and status code. As a good programming practice, it’s always required that server response should contain accurate headers and status code. Without ResponseEntity, you can control the response body but it will be hard to control the headers and status code.

**6) What is a use of devtools?**

**7) Different embedded servers supported by Spring boot? Which is default one? and how we can use another embedded server?**

**🡪**  Spring Boot supports Tomcat, Undertow, and Jetty as embedded servers. Default embedded server is Tomcat.

server.undertow.enabled: true;

[Apache Tomcat](https://tomcat.apache.org/) and [Eclipse Jetty](https://www.eclipse.org/jetty/) are two of the most popular web servers and Java Servlet containers. Tomcat is more widely used compared to Jetty and has significantly more market share. On the other hand, Jetty is light-weight, more compact and has a smaller CPU and memory footprint. For this reason, it is easier to work with it in development than Tomcat. This is not to suggest that Jetty isn’t good for production - in my experience, it is as performant as Tomcat if not more.

Many developers prefer Jetty over Tomcat during the development stage when they want to rapidly launch and test web apps on their local machines. **Spring Boot**[**web starter**](https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web) uses Tomcat as the default embedded server. Let’s take a look at how to change it to Jetty.

1. Exclude Tomcat from web starter dependency, since it is added by default
2. Add the Jetty dependency

**8) Different ways of creating spring boot application?**

**🡪 1. Using spring initializ**r and import in STS, Eclipse, IntelliJ idea, VSCode, Netbeans

**2. Using Spring Starter Project** in STS (Eclipse)

**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.

**9) Explain SpringApplication.run(...) method.**

**🡪** It is present under package org.springframework.boot. In startup process after the context is initialized, spring boot calls its run () method with command-line arguments provided to the application.

**10) What are the advantages of starter poms?**

**🡪** Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project.

The dependency of spring Boot starter is the parent pom which provides plugin management and dependency for supporting the applications which are based on Spring Boot. Spring Boot starter is having the Default Java version for the purpose of configuration. The parent pom allows you to manage different things for performing child projects like configuration, dependency management and Default plugin configuration. Pom is a parent declaration of spring Boot starter and without it the dependencies in the project can not be defined. To omit the version number of spring Boot you can import some additional starters but be cautious while doing the same

**\*\* Dependency Management** is just a way of managing all the required dependencies in one place and efficiently making use of them. It allows specifying all required dependencies according to the respective Spring-Boot version in one place itself.

**11) What is content negotiation and how we can achieve it in Spring boot?**

**12) Different ways of handling exceptions in Sprinig boot? => You suppose to explain @ExceptionHandler and @ControllerAdvice**

**13) How you document your REST APIs? => You suppose to explain about swagger**

**14) What is spring boot actuator? Can we create custom endpoint and how?**

**🡪**

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

**Spring Boot Actuator**  help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use **HTTP** and **JMX** endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the S**pring Boot actuator.**

**Endpoint:** The actuator endpoints allows us to monitor and interact with the application. Spring Boot provides a number of built-in endpoints. We can also create our own endpoint. We can enable and disable each endpoint individually. Most of the application choose **HTTP**, where the Id of the endpoint, along with the prefix of **/actuator,**is mapped to a URL.

For example, the **/health** endpoint provides the basic health information of an application. The actuator, by default, mapped it to **/actuator/health**.

There are **three** main features of Spring Boot Actuator:

* **Endpoints Metrics** **Audit**

13 default endpoints are there in actuator endpoint.

**15) How to enable cors in spring boot?**

**16) What is Spring Data JPA? How we can define our custom queries?**

**17) What Are Possible Sources of External Configuration?**

**18) Which Is a Better Way to Configure a Spring Boot Project – Using Properties or YAML? [ What are advantages of yaml over proeprties file OR difference between yaml and properties file ]**

**🡪**

| **yaml** | **Properties** |
| --- | --- |
| Contains key and value pairs | Key and values separated by equal or color |
| Human readable format | Easy to read by humans |
| Supports Integer, Strings, Maps, and Lists | Supports primitive types like strings and numbers |
| Supports hierarchical structure | Supports flat and non-hierarchical structure |

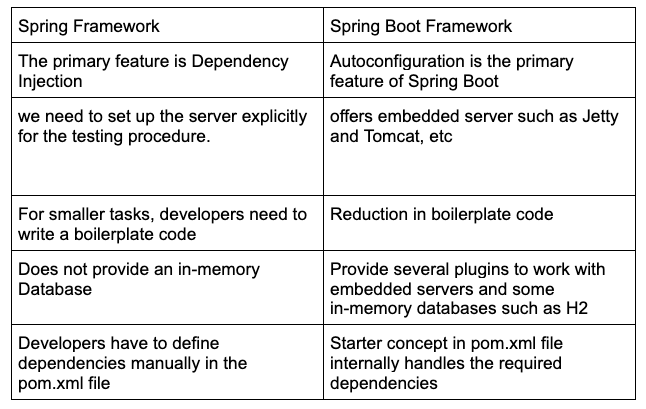
| YAML(.yml) | .properties |
| --- | --- |
| Spec can be found [here](https://yaml.org/spec/) | It doesn’t really actually have a spec. The closest thing it has to a spec is actually the javadoc. |
| Human Readable (both do quite well in human readability) | Human Readable |
| Supports key/val, basically map, List and scalar types (int, string etc.) | Supports key/val, but doesn’t support values beyond the string |
| Its usage is quite prevalent in many languages like Python, Ruby, and Java | It is primarily used in java |
| Hierarchical Structure | Non-Hierarchical Structure |
| Spring Framework doesn’t support @PropertySources with .yml files | supports @PropertySources with .properties file |
| If you are using spring profiles, you can have multiple profiles in one single .yml file | Each profile need one separate .properties file |
| While retrieving the values from .yml file we get the value as whatever the respective type (int, string etc.) is in the configuration | While in case of the .properties files we get strings regardless of what the actual value type is in the configuration |

**What should I use .properties or .yml file?**

Strictly speaking, .yml file is advantageous over .properties file as it has type safety, hierarchy and supports list but if you are using spring, spring has a number of conventions as well as type conversions that allow you to get effectively all of these same features that YAML provides for you.

One advantage that you may see out of using the YAML(.yml) file is if you are using more than one application that read the same configuration file. you may see better support in other languages for YAML(.yml) as opposed to .properties.

**19) Spring framework vs Spring boot.**



**Q4. What Is Spring Initializr?**

Spring Initializr is a convenient way to create a Spring Boot 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.

## **\*\* 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 the 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.

**3) What is autoconfiguration and how does Spring boot does auto-configuration? How to Disable a Specific Auto-Configuration? How to Register a Custom Auto-Configuration?**

## **🡪 Spring Boot Auto Configuration**

All auto-configuration logic is implemented in spring-boot-autoconfigure.jar. Spring-based application requires a lot of configuration. When we use Spring MVC, we need to configure dispatcher servlet, view resolver, web jars among other things.

**and how does Spring boot does auto-configuration?-->** Spring Boot detects classes in the classpath and auto-configuration mechanism will ensure to create and wires necessary beans for us. This is one of the most powerful feature of the Spring Boot and most of the work happens silently in the background.

**Auto-configuration of Spring Application Context automatically configures the beans which are likely to be needed in application development.**

**We use**@EnableAutoConfiguration**annotation for auto-configuration.** We generally use @SpringBootApplication annotation in a Spring application, and this annotation enables the auto-configuration of the context automatically. So there is no additional use of using @EnableAutoConfiguration with @SpringBootApplication.

Auto-configuration classes are regular Spring @Configuration beans. Auto-configuration does help in configuration of the context and get us started quickly. Spring Boot tries to auto-configure the application context based on jars we provide in the classpath. For example, if HSQLDB dependency is in the classpath, Spring Boot will configure an in-memory database (if we don’t configure database connection beans manually).

## 2. Disabling auto-configuration in Spring Boot

We’ll see two ways to disable auto-configuration in Spring Boot:

* Disable using annotation
* Disable using property file

Disabling selective classes is very helpful while testing.

### 2.1 Disable auto-configuration using annotation

We can use exclude attribute of @SpringBootApplication to disable auto-configuration. For example, following code disables auto-configuration for Spring Data JPA:

@SpringBootApplication(exclude = {

DataSourceAutoConfiguration.class,

DataSourceTransactionManagerAutoConfiguration.class,

HibernateJpaAutoConfiguration.class

})

If the class is not on the classpath, we can use the excludeName attribute. We have to provide fully qualified class name in this case.

### 2.2 Disable using property file

If we don’t want to use annotation to exclude files, we can use spring.autoconfigure.exclude property. For example, following code disables auto-configuration for Spring Data JPA using spring.autoconfigure.exclude property.

spring.autoconfigure.exclude= \

org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration, \

org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration, \

org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration

Spring Boot auto-configuration automatically configures the Spring application based on the jar dependencies that we have added.

For example, if the H2 database Jar is present in the classpath and we have not configured any beans related to the database manually, the Spring Boot's auto-configuration feature automatically configures it in the project.

We can enable the auto-configuration feature by using the annotation **@EnableAutoConfiguration**. But this annotation does not use because it is wrapped inside the **@SpringBootApplication** annotation. The annotation @SpringBootApplication is the combination of three annotations: **@ComponentScan, @EnableAutoConfiguration,** and **@Configuration**. However, we use @SpringBootApplication annotation instead of using @EnableAutoConfiguration.

**@SpringBootApplication=@ComponentScan+@EnableAutoConfiguration+@Con**x

When we add the **spring-boot-starter-web** dependency in the project, Spring Boot auto-configuration looks for the Spring MVC is on the classpath. It auto-configures **dispatcherServlet**, a default **error page,** and **web jars**.

Similarly, when we add the **spring-boot-starter-data-jpa** dependency, we see that Spring Boot Auto-configuration, auto-configures a **datasource** and an **Entity Manager**.

All auto-configuration logic is implemented in **spring-boot-autoconfigure.jar**, as shown in the following figure.

## **Need of auto-configuration**

Spring-based application requires a lot of configuration. When we use Spring MVC, we need to configure **dispatcher servlet, view resolver, web jars** among other things. The following code shows typical configuration of a dispatcher servlet in a web application:

1. **<servlet>**
2. **<servlet-name>**dispatcher**</servlet-name>**
3. **<servlet-class>**
4. org.springframework.web.servlet.DispatcherServlet
5. **</servlet-class>**
6. **<init-param>**
7. **<param-name>**contextConfigLocation**</param-name>**
8. **<param-value>**/WEB-INF/todo-servlet.xml**</param-value>**
9. **</init-param>**
10. **<load-on-startup>**1**</load-on-startup>**
11. **</servlet>**
12. **<servlet-mapping>**
13. **<servlet-name>**dispatcher**</servlet-name>**
14. **<url-pattern>**/**</url-pattern>**
15. **</servlet-mapping>**

Similarly, when we use Hibernate/ JPA, we need to configure datasource, a transaction manager, an entity manager factory among a host of other things.

**Configuring datasource**

1. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
2. destroy-method="close"**>**
3. **<property** name="driverClass" value="${db.driver}" **/>**
4. **<property** name="jdbcUrl" value="${db.url}" **/>**
5. **<property** name="user" value="${db.username}" **/>**
6. **<property** name="password" value="${db.password}" **/>**
7. **</bean>**
8. **<jdbc:initialize-database** data-source="dataSource"**>**
9. **<jdbc:script** location="classpath:config/schema.sql" **/>**
10. **<jdbc:script** location="classpath:config/data.sql" **/>**
11. **</jdbc:initialize-database>**

**Configuring entity manager factory**

1. **<bean**
2. class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
3. id="entityManagerFactory"**>**
4. **<property** name="persistenceUnitName" value="hsql\_pu" **/>**
5. **<property** name="dataSource" ref="dataSource" **/>**
6. **</bean>**

**Configuring transaction manager**

1. **<bean** id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"**>**
2. **<property** name="entityManagerFactory" ref="entityManagerFactory" **/>**
3. **<property** name="dataSource" ref="dataSource" **/>**
4. **</bean>**
5. **<tx:annotation-driven** transaction-manager="transactionManager"**/>**

## **Disable Auto-configuration Classes**

We can also disable the specific auto-configuration classes, if we do not want to be applied. We use the **exclude** attribute of the annotation @EnableAutoConfiguration to disable the auto-configuration classes. For example:

1. import org.springframework.boot.autoconfigure.\*;
2. import org.springframework.boot.autoconfigure.jdbc.\*;
3. import org.springframework.context.annotation.\*;
4. @Configuration(proxyBeanMethods = false)
5. @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
6. public class MyConfiguration
7. {
8. }

We can use the attribute **excludeName** of the annotation @EnableAutoConfiguration and specify the **qualified** name of the class, if the class is not on the class path. We can exclude any number of auto-configuration classes by using the property **spring.autoconfigure.exclude**.