**Spring Boot Interview Questions**

**Q1. What is Spring Boot and What Are Its Main Features?**

Spring Boot is a framework for rapid application development built on top of the Spring Framework. With its auto-configuration and embedded application server support.

Spring Boot is one of the most popular technologies in the Java ecosystem as of date.

Here are a few salient features:

* **[Starters](https://www.baeldung.com/spring-boot-starters)**– a set of dependency descriptors to include relevant dependencies at a go
* **[Auto-configuration](https://www.baeldung.com/spring-boot-annotations" \l "enable-autoconfiguration)** – a way to automatically configure an application based on the dependencies present on the classpath
* **[Actuator](https://www.baeldung.com/spring-boot-actuators)**– to get production-ready features such as monitoring the application.
* **[Security](https://www.baeldung.com/security-spring)**
* **[Logging](https://www.baeldung.com/spring-boot-logging)**

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

**Spring :**

> Spring Framework is a widely used Java EE framework for building applications. Spring is an open-source lightweight framework widely used to develop enterprise applications.

> It aims to simplify Java EE development that makes developers more productive.

> The primary feature of the Spring Framework is dependency injection.

> It helps to make things simpler by allowing us to develop loosely coupled applications.It helps to create a loosely coupled application.

> The developer writes a lot of code (boilerplate code) to do the minimal task.To create a Spring application, the developers write lots of code.

> To test the Spring project, we need to set up the sever explicitly.To run the Spring application, we need to set the server explicitly.

> It does not provide support for an in-memory database.It doesn’t provide support for the in-memory database.

> Developers manually define dependencies for the Spring project in pom.xml.To run the Spring application, a deployment descriptor is required.

**Spring Boot :**

> Spring Boot Framework is widely used to develop REST APIs.

> It aims to shorten the code length and provide the easiest way to develop Web Applications.

> The primary feature of Spring Boot is Autoconfiguration. It automatically configures the classes based on the requirement.

> It helps to create a stand-alone application with less configuration.

> It reduces boilerplate code.

> Spring Boot offers embedded server such as Jetty and Tomcat, etc.

> It offers several plugins for working with an embedded and in-memory database such as H2.

> Spring Boot comes with the concept of starter in pom.xml file that internally takes care of downloading the dependencies JARs based on Spring Boot Requirement.

> Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.

> The most important feature of the Spring Boot is Autoconfiguration.

> It helps to create a stand-alone application.

> Spring Boot provides embedded servers such as Tomcat and Jetty etc.

> There is no requirement for a deployment descriptor.

> It reduces the lines of code.

> It provides support for the in-memory database such as H2.

**Spring Boot** and **Spring MVC** exist for different purposes. The primary comparison between Spring Boot and Spring MVC are discussed below:

**Spring Boot**

> Spring Boot is a module of Spring for packaging the Spring-based application with sensible defaults.

> It provides default configurations to build Spring-powered framework.

> There is no need to build configuration manually.

> There is no requirement for a deployment descriptor. Web. xml, also known as deployment descriptor, is traditionally used as a configuration file for Java web applications. It defines servlets, their mappings, servlet filters, lifecycle listeners and more. ... With Servlet version 3.0, the deployment descriptor is no longer mandatory.

> It avoids boilerplate code and wraps dependencies together in a single unit.

> It reduces development time and increases productivity.

**Spring MVC**

> Spring MVC is a model view controller-based web framework under the Spring framework.

> It provides ready to use features for building a web application.

> It requires build configuration manually.

> A Deployment descriptor is required.

> It specifies each dependency separately.

> It takes more time for development time & increase productivity.

|  |  |
| --- | --- |
| **Spring** | **Spring boot** |
| the Spring framework focuses on providing flexibility through its dependency injection feature. It helps to inject the required dependencies quickly but also to develop your application in a loosely coupled fashion | Autoconfiguration: Developers can automatically configure their Spring application. However, Spring Boot is also capable of changing the configuration based on the dependencies you list. For example, when you list “MySQL” as a dependency, it will configure your Spring application with the “MySQL connector” included. And if you want to add a custom configuration, you can create a class that overrides the default configuration for your “MySQL connector”. |
| A lightweight framework. | Standalone: There’s no need to deploy your application to a web server. You simply enter the run command to start the application. |
| Helps with loose coupling dependencies and testability. The modular architecture allows you to pick the parts you need and isolate them. | Opinionated: On the [official page](https://spring.io/projects/spring-boot" \t "https://stackify.com/what-is-spring-boot/_blank), we find that Spring Boot decides for you which defaults to use for the configuration. Also, it decides which packages to install for the dependencies you require. For example, if you include the Spring Boot starter “pom” for “JPA”, it will autoconfigure an in-memory database, a hibernate entity manager, and a simple data source. This is an example of an opinionated default configuration that you can override. While some developers might feel this is too opinionated, Spring Boot’s opinionated setup helps developers to get started quickly on their projects. |
| Has support for both XML and annotation configuration. |  |
|  |  |
|  |  |
|  |  |

### **Why Is Spring Boot So Popular?**

First of all, it makes use of [Java](https://stackify.com/java-tutorials/), which is one of the most popular programming languages in the world. Besides that, Spring Boot is an amazing tool that helps you to get enterprise-grade applications up and running quickly without having to worry about configuring your application correctly and safely.

Also, the user community is huge. If you want free learning materials and courses, you’ll find plenty out there. The accessibility of education has had a big impact on the framework’s popularity.

### **Some additional benefits include :**

* Reduces development time and increases the overall productivity of the development team.
* Helps you autoconfigure all components for a production-grade Spring application.
* Makes it easier for developers to create and test Java-based applications by providing a default setup for unit and integration tests.
* Avoids writing lots of boilerplate code, annotations, and XML configuration.
* Comes with embedded HTTP servers like [Tomcat or Jetty](https://stackify.com/tomcat-vs-jetty-vs-glassfish-vs-wildfly/) to test web applications.
* Adds many plugins that developers can use to work with embedded and in-memory databases easily. Spring allows you to easily connect with database and queue services like Oracle, PostgreSQL, MySQL, MongoDB, Redis, Solr, ElasticSearch, Rabbit MQ, ActiveMQ, and [many more](https://spring.io/guides" \t "https://stackify.com/what-is-spring-boot/_blank).
* Allows admin support—meaning you can manage via remote access to the application.

## **What is Spring Boot?**

Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can just run. You can get started with minimum configurations without the need for an entire Spring configuration setup.

### **Advantages :**

Spring Boot offers the following advantages to its developers −

* Easy to understand and develop spring applications
* Increases productivity
* Reduces the development time

### **Goals**

Spring Boot is designed with the following goals −

* To avoid complex XML configuration in Spring
* To develop a production ready Spring applications in an easier way
* To reduce the development time and run the application independently
* Offer an easier way of getting started with the application

## **Why Spring Boot?**

You can choose Spring Boot because of the features and benefits it offers as given here −

It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.

It provides a powerful batch processing and manages REST endpoints.

In Spring Boot, everything is auto configured; no manual configurations are needed.

It offers annotation-based spring application

Eases dependency management

It includes Embedded Servlet Container.

# **How Spring Boot Application Works Internally ?**

**Spring Boot Application Internal Working :** Spring does not generate any code automatically and not using any xml configuration file . so spring uses internally pragmatically configuration done by spring boot developer that are provided by jar.  
we are using just pre-configured jar . and those jar available in: **META-INF/spring.factories**

Enable  
Disable

To Enable preconfigured jars we just need to define dependency in pom.xml file.

<dependency>

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

This dependency will load all the jars related to JPA repository and stored into spring.factories.  
you can go to maven dependencies then click and open spring-boot-autoconfigure jar in the last you will see META-INF folder inside this spring.factories here you will find your jar org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration.

Based on **@Conditional** and**@Configuration** :

*****@Configuration******(proxyBeanMethods = false)******@ConditionalOnBean******(DataSource.class)******@ConditionalOnClass******(JpaRepository.class)******@ConditionalOnMissingBean******({ JpaRepositoryFactoryBean.class, JpaRepositoryConfigExtension.class })******@ConditionalOnProperty******(prefix = “spring.data.jpa.repositories”, name = “enabled”, havingValue = “true”,  
matchIfMissing = true)******@Import******(JpaRepositoriesRegistrar.class)******@AutoConfigureAfter******({ HibernateJpaAutoConfiguration.class, TaskExecutionAutoConfiguration.class })  
public class JpaRepositoriesAutoConfiguration {}*

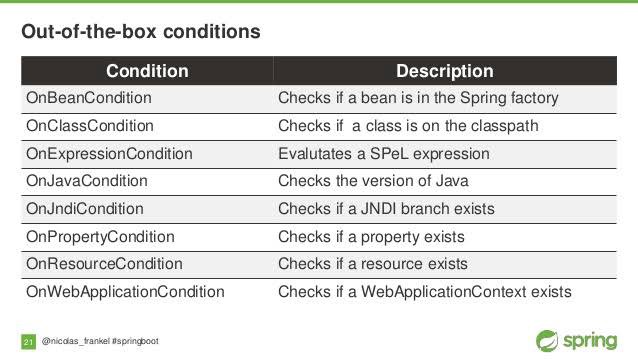
**@ConditionalOnBean(DataSource.class) :**— — — — — — — — — — — — — — —  
It will serach for the DataSource bean if it is available then only it will enable JpaRepositoriesAutoConfiguration . So this we need to define DataSource related properties into our property file.

**@ConditionalOnClass(JpaRepository.class) :**— — — — — — — — — — — — — — —  
It will serach for the JpaRepository class if it is available then only it will enable JpaRepositoriesAutoConfiguration .

like this :  
**@ConditionalOnMissingBean**({ JpaRepositoryFactoryBean.class, JpaRepositoryConfigExtension.class })  
**@ConditionalOnProperty**(prefix = “spring.data.jpa.repositories”, name = “enabled”, havingValue = “true”, matchIfMissing = true)

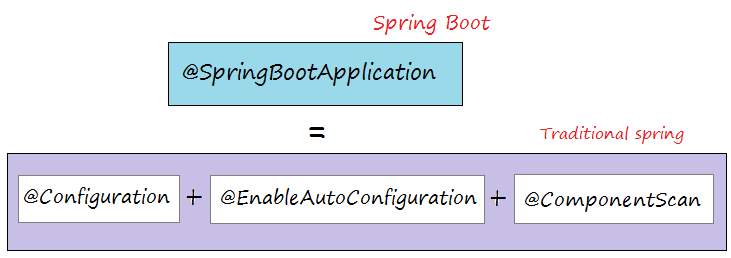
If all conditions are true then only it will enable JpaRepositoriesAutoConfiguration class.

**The mainly Conditions checked by spring boot :**



If all the conditions are satisfied then only spring will enable to the component.

**@SpringBootApplication**is the main annotation that we used on our main method and this annotation is the combination of these three annotations :



**High Level Flow Of Spring Boot And How run Method works :** From the run method, the main application context is kicked off which in turn searches for the classes annotated with @Configuration, initializes all the declared beans in those configuration classes, and based upon the scope of those beans, stores those beans in JVM, specifically in a space inside JVM which is known as IOC container. After the creation of all the beans, automatically configures the dispatcher servlet and registers the default handler mappings, messageConverts, and all other basic things.

Basically, spring boot supports three embedded servers:- Tomcat (default), Jetty and Undertow.

run() internal flow :  
==========

1. create application context  
2. check Application Type  
3. Register the annotated class beans with the context  
4. Creates an instance of TomcatEmbeddedServletContainer : and adds the context. Used to deploy our jar automatically.

**open SpringApplication.class :**And find here run(String… args) method inside this method you will see the method **createApplicationContext**() so first it will create application context and inside createApplicationContext() method it will check application type it is SERVLET type Or REACTIVE or DEFAULT context type based on this it will return context. Now in DEFAULT\_CONTEXT\_CLASS you will see the class **AnnotationConfigApplicationContext**.class.

*public******AnnotationConfigApplicationContext******(Class… annotatedClasses) {  
this();  
register(annotatedClasses);  
refresh();  
}*

open this class its constructor is used to Register the annotated class beans with the context. The classes which are annotated with **@Component, @Service, @Configuration** etc. will be register to the context. And in the finally run(-) method auto deploy the jar/war to server.

**@Configuration :**It will behave act as bean.  
**[@EnableAutoConfiguart](http://twitter.com/EnableAutoConfiguart)ion :**it will enable bean based on some condition that we have discussed above.  
**[@ComponentScan](http://twitter.com/ComponentScan) :**It is mainly used to scan the classes and packages to create the bean.

It is the main class that we need to define to make our spring boot application.

***[@SpringBootApplicatio](http://twitter.com/SpringBootApplicatio)****n***** *public class Application******{***** *public static void main(String[] args) {  
SpringApplication.run(Application.class, args);  
}******}*****

If we will open @SprinBootApplication Annotation here you will see it contains :

[@SpringBootConfigurat](http://twitter.com/SpringBootConfigurat)ion  
[@EnableAutoConfigurat](http://twitter.com/EnableAutoConfigurat)ion  
[@ComponentScan](http://twitter.com/ComponentScan)(excludeFilters = {  
[@Filter](http://twitter.com/Filter)(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),  
[@Filter](http://twitter.com/Filter)(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })  
public [@interface](http://twitter.com/interface) **SpringBootApplication** {  
// code here…..  
}

**Application Bootstrap:**

The basic **difference** in bootstrapping of an application in **Spring** and **Spring Boot** lies with the servlet. Spring uses either the **web.xml** or **SpringServletContainerInitializer** as its bootstrap entry point.

On the other hand, Spring Boot uses only Servlet 3 features to bootstrap an application.

**How Spring Bootstraps? (how spring works)?**

Spring supports both the legacy **web.xml** way of bootstrapping as well as the latest **Servlet 3+** method.

**Let's see the web.xml approach in steps:**

> Servlet container (the server) reads web.xml

> The DispatcherServlet defined in the web.xml is instantiated by the container

> DispatcherServlet creates WebApplicationContext by reading WEB-INF/{servletName}-servlet.xml

> Finally, the DispatcherServlet registers the beans defined in the application context.

**Here's how Spring bootstraps using Servlet 3+ approach:**

> The container searches for classes implementing ServletContainerInitializer and executes.

> The SpringServletContainerInitializer finds all classes implementing WebApplicationInitializer

> The WebApplicationInitializer creates the context with XML or @Configuration classes.

> The WebApplicationInitializer creates the DispatcherServlet with the previously created context.

**How Spring Boot Bootstraps? (how spring Boot works)?**

The entry point of a Spring Boot application is the class which is annotated with **@SpringBootApplication:**

**@SpringBootApplication**

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

By default, Spring Boot uses an embedded container to run the application. Spring Boot uses the **public static void main** entry-point to launch an embedded web server.

Also, it takes care of the binding of the Servlet, Filter, and ServletContextInitializer beans from the application context to the embedded servlet container.

Another feature of Spring Boot is that it automatically scans all the classes in the same package or sub packages of the Main-class for components.

Spring Boot provides the option of deploying it as a web archive in an external container as well. In this case, we have to extend the **SpringBootServletInitializer**:

**@SpringBootApplication**

public class Application extends **SpringBootServletInitializer** {

// ...

}

Here the external servlet container looks for the Main-class defined in the META-INF file of the web archive and the SpringBootServletInitializer will take care of binding the Servlet, Filter, and ServletContextInitializer.

**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.

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>**

**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?**

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

**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.

// 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.DataSourceAutoConfiguration

**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.CustomAutoConfiguration

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.

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.

**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.

**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.

**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

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.

**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.

**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.

**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.

**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.active=dev**

To set profiles programmatically, we can also use the SpringApplication class:

SpringApplication.setAdditionalProfiles("dev");

To set profiles using Maven in Spring Boot, we can specify profile names under spring-boot-maven-plugin in pom.xml:

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<profiles>

<profile>dev</profile>

</profiles>

</configuration>

</plugin>

...

</plugins>

and execute the Spring Boot-specific Maven goal:

**mvn spring-boot:run**

**Q: Can we disable the default web server in the Spring Boot application?**

The major strong point in Spring is to provide flexibility to build your application loosely coupled. Spring provides features to disable the web server in a quick configuration. **Yes,** we can use the application.properties to configure the web application type, i.e. **spring.main.web-application-type=none.**

**Q: Can we override or replace the Embedded Tomcat server in Spring Boot?**

Yes, we can replace the Embedded Tomcat with any other servers by using the Starter dependencies. You can use spring-boot-starter-jetty or spring-boot-starter-undertow as a dependency for each project as you need.

**Q: Is this possible to change the port of Embedded Tomcat server in Spring boot?**

Yes, it's possible to change the port. You can use the application.properties file to change the port. But you need to mention "server.port" (i.e. server.port=8081). Make sure you have application.properties in your project classpath; REST Spring framework will take care of the rest. If you mention server.port=0 , then it will automatically assign any available port.

**Q: What is a shutdown in the actuator?**

Shutdown is an endpoint that allows the application to be gracefully shutdown. This feature is not enabled by default. You can enable this by using management.endpoint.shutdown.enabled=true in your application.properties file. But be careful about this if you are using this.

**Q: What is the Spring Initializer?**

The Spring Initializer is a web application that generates a Spring Boot project with everything you need to start it quickly. As always, we need a good skeleton of the project; it helps you to create a project structure/skeleton properly.

**Q: How to enable/disable the Actuator?**

Enabling/disabling the actuator is easy; the simplest way is to enable features to add the dependency (Maven/Gradle) to the spring-boot-starter-actuator, i.e. Starter. If you don't want the actuator to be enabled, then don't add the dependency.

**Q: What is Spring Actuator? What are its advantages?**

"An actuator is a manufacturing term that refers to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change."

As we know, Spring Boot provides lots of auto-configuration features that help developers quickly develop production components. But if you think about debugging and how to debug, if something goes wrong, we always need to analyze the logs and dig through the data flow of our application to check to see what's going on. So, the Spring Actuator provides easy access to those kinds of features. It provides many features, i.e. what beans are created, the mapping in the controller, the CPU usage, etc. Automatically gathering and auditing health and metrics can then be applied to your application.

It provides a very easy way to access the few production-ready REST endpoints and fetch all kinds of information from the web. But by using these endpoints, you can do many things to see here the endpoint docs. There is no need to worry about security; if Spring Security is present, then these endpoints are secured by default using Spring Security’s content-negotiation strategy. Or else, we can configure custom security by the help of RequestMatcher.

**Q: How to disable a specific auto-configuration class?**

You can use the exclude attribute of@EnableAutoConfiguration, if you find any specific auto-configuration classes that you do not want are being applied.

**By using "exclude" @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})**

On the other foot, if the class is not on the classpath, you can use the excludeName attribute of the annotation and specify the fully qualified name instead.

//By using "excludeName"

**@EnableAutoConfiguration(excludeName={Foo.class})**

Also, Spring Boot provides the facility to control the list of auto-configuration classes to exclude by using the spring.autoconfigure.exclude property. You can add into the application.properties. And you can add multiple classes with comma separated.

//By using property file

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

**Q: How to exclude any package without using the basePackages filter?**

There are different ways you can filter any package. But Spring Boot provides a trickier option for achieving this without touching the component scan. You can use the exclude attribute while using the annotation @SpringBootApplication. See the following code snippet:

@SpringBootApplication(exclude= {Employee.class})

public class FooAppConfiguration {}

**Q: What does the @SpringBootApplication annotation do internally?**

As per the Spring Boot doc, the @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes. Spring Boot enables the developer to use a single annotation instead of using multiple. But, as we know, Spring provided loosely coupled features that we can use for each individual annotation as per our project needs.

**Q: Can Spring Boot also be used to create non-web applications?**

Yes, Spring Boot supports the development of both web and non-web applications. We need to remove web dependencies from the classpath and the application context to create a non-web application.

**Q: Can you disable particular auto-configuration in spring boot? Explain how?**

Yes, we can do that by

Using the exclude attribute of **@EnableAutoConfiguration**

@Configuration

**@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})**

public class CustomConfiguration {}

using the exclude attribute for @SpringBootApplication annotation

@SpringBootApplication(exclude= DataSourceAutoConfiguration.class)

public class CustomApplication {}

**Q: Describe spring-boot-starter-parent?**

It is a unique starter which adds jars to the classpath for easy Maven or Gradle dependency management.

**Q: How and where can you define properties in Spring Boot?**

We can use the application.properties file to define features. This file gets automatically downloaded by Spring Boot and is used for application configuration.

**Q: What do you understand by the shutdown in the actuator?**

It is a management endpoint that allows for a smooth and proper shutdown of an application. It is not authorized by default, but it can be enabled by using **management.endpoint.shutdown.enabled=true.**

**Q:What embedded containers are supported by Spring Boot?**

Spring Boot supports three embedded containers:

Tomcat (used by default)

Undertow

Jetty

**Q: What is Thymeleaf in Spring Boot?**

It is a Java-based server-side template engine that offers elegant and natural templates for a web application.

**Q: Can you name and briefly explain all the spring boot** components/features?

**The main features of Spring Boot are :**

**1. Starter dependency**

There are many dependencies in the Spring framework, and this feature aggregates them together. Spring Boot comes packed with several starter dependencies to enhance productivity.

Few of the Spring Boot Starters are Test Starter, Web Starter, Mail Starter, and more. For instance, if we want to use JPA and Spring for database access, we can add this starter dependency in the project-spring-boot-starter-data-jpa.

**2. Auto-Configuration**

This feature scans the classpath and searches the libraries/Jars in the classpath to provide the necessary configuration to design and run the application.

For example, while developing an application with Spring Boot, if there is a Thymeleaf.jar present in the classpath, it automatically can align the Thymeleaf template resolver and other settings.

**3. Spring Initializer**

It is a web application that simplifies the process of the project set up by creating the initial project structure and build scripts. It increases productivity by reducing development time.

**4. Spring Actuator**

Actuators are incredibly significant for microservices as they enable deployment-ready features like auditing, health check-up, log information, etc. for running Spring boot applications. Actuators have built-in management endpoints that are secured by default.

For example:

/beans– it exhibits the entire list of all Spring beans in your application.

/health— it displays application health information by monitoring the production system

**5. Spring CLI**

This feature allows the developers to use Groovy for writing the Spring boot application, hence resulting in a more concise code.

**Q: What is the need for Spring Boot?**

While Spring offers developers an ideal environment to develop large applications, the amount of configuration and its complexity makes it challenging to do so.

Here is where Spring Boot comes to rescue. Its features like pre-built templates and auto-configuration allow developers to use existing spring functionalities with more ease, minimum effort, and maximum efficiency.

**The main advantages of Spring Boot are:**

> It reduces development and testing time.

> It uses JavaConfig instead of XML.

> It provides an opinionated development approach.

> It offers starter projects or defaults for agile development.

> No separate web server is required; hence there is no need to boot up Glassfish, Tomcat, or any other server.

> Reduce Developement, Testing time and efforts.

> Use of JavaConfig helps avoid usage of XML.

> Avoid lots of maven imports and the various version conflicts.

> Provide Opinionated Development approach.

> Quick start to development by providing defaults.

> No Separate Web Server Needed.Which means that you no longer have to boot up Tomcat, Glassfish, or anything else.

> Requires less configuration-Since there is no web.xml file. Simply add classes annotated with@Configuration and then you can add methods annotated with@Bean, and Spring will automagically load up the object and manage it like it always has. You can even add @Autowired to the bean method to have Spring autowire in dependencies needed for the bean.

> Environment Based Configuration-Using these properties, you can pass into the application which environment you are using with:-Dspring.profiles.active={enviornment}. Spring will then load up the subsequent application properties file at (application-{environment}.properties) after loading up the main application properties file.

**Q: What is Spring Boot, and how is it different from Spring?**

It takes the Java development with Spring to the next level and simplifies the development by removing major pain points associated with dependency, configuration, and management. The differences are listed below.

**Spring**

It is a Java-based web application framework.

It is quite complicated, making it difficult to use.

It offers libraries and tools to create customized web applications.

It takes an un-opinionated view.

It requires XML configurations.

**Spring Boot**

It is a module of Spring used for creating microservices.

It is less complicated and more robust as compared to the spring framework.

It creates stand-alone spring application projects that can run/ execute.

It takes an opinionated view as it can decide which defaults to deploy the configuration.

It does not require XML configurations.

**Q: What is JavaConfig?**

Spring **JavaConfig** is a product of the Spring community that provides a pure-Java approach to configuring the Spring IoC Container. It helps avoid using XML configurations.

The advantages of JavaConfig are:

**Object-oriented configuration.** Because configurations are defined as classes in JavaConfig, users can take full advantage of object-oriented features in Java. One configuration class may subclass another, overriding its @Bean methods, etc.

**Reduced or eliminated XML configuration.** The benefits of externalized configuration based on the principles of dependency injection have been proven.

JavaConfig provides developers with a pure-Java approach to configuring the Spring container that is conceptually similar to XML configuration. It is technically possible to configure the container using only JavaConfig configuration classes, however in practice many have found it ideal to mix-and-match JavaConfig with XML.

**Type-safe and refactoring-friendly**. JavaConfig provides a type-safe approach to configuring the Spring container. Thanks to Java 5.0's support for generics, it is now possible to retrieve beans by type rather than by name, free of any casting or string-based lookups.

**Q: How to reload my changes on Spring Boot without having to restart server?**

This can be achieved using DEV Tools. With this dependency any changes you save, the embedded tomcat will restart. Spring Boot has a Developer tools (DevTools) module which helps to improve the productivity of developers.

One of the key challenge for the Java developers is to auto deploy the file changes to server and auto restart the server. Developers can reload changes on Spring Boot without having to restart my server. This will eliminates the need for manually deploying the changes every time. Spring Boot doesnt have this feature when it has released its first version. This was a most requested features for the developers. The module DevTools does exactly what is needed for the developers. This module will be disabled in the production environment. It also provides H2-database console for better testing the application. The following dependency is used

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-devtools</artifactId>**

**<optional>true</optional>**

**</dependency>**

The DevTool dependency usage for autorestart and H2 DB console.

**Q: What is Actuator in Spring Boot?**

Spring boot actuator is one of the important feature in spring boot framework. Spring boot actuator helps you to access the current state of the running application in production environment. There are several metrics that has to be checked and monitored in the production environment. Even some external applications may be using those services to trigger the alert message to concerned person. Actuator module exposes set of REST endpoints that can be directly accessed as a HTTP URL to check the status.

**Q: How to implement JWT authentication for Spring Boot Application?**

JWT stands for JSON Web Token. JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. The client will need to authenticate with the server using the credentials only once. During this time the server validates the credentials and returns the client a JSON Web Token(JWT). For all future requests the client can authenticate itself to the server using this JSON Web Token(JWT) and so does not need to send the credentials like username and password.

**Q:How to disable Actuator endpoint security in Spring Boot?**

By default all sensitive HTTP endpoints are secured such that only users that have an ACTUATOR role may access them. Security is enforced using the standard HttpServletRequest.isUserInRole method.

We can disable security using -

management.security.enabled=false

It is suggested to disable security only if the actuator endpoints are accessed behind firewall.

**Q: How to run Spring boot application to custom port?**

In order to run a spring boot application on a custom port you can specify the port in application.properties.

server.port=8090

**Q: What is YAML?**

YAML is a human-readable data serialization language. It is commonly used for configuration files.

Compared to properties file, YAML file is much more structured and less confusing in case we want to add complex properties in the configuration file. As can be seen YAML has hierarchical configuration data.

**Q: How to implement security for Spring boot application?**

For Implementing security for Spring Boot we use the spring-boot-starter-security dependency and have to add the Security config. It requires very little code. Config class will have to extend WebSecurityConfigurerAdapter and override its methods.

**Q: Have you integrated Spring Boot and ActiveMQ?**

For integrating Spring Boot and ActiveMQ we use the spring-boot-starter-activemq dependency

It requires very little configuration and no boilerplate code.

**Q: Have you integrated Spring Boot and Apache Kafka?**

For integrating Spring Boot and Apache Kafka we use the spring-kafka dependency.

**Q: How to implement Pagination and Sorting with Spring Boot?**

Using Spring Boot achieving pagination is very simple. Using the Spring Data-JPA this is achieved passing pageable org.springframework.data.domain.Pageable to the repository methods.

**Q: What is Swagger? Have you implemented it using Spring Boot?**

Swagger is widely used for visualizing APIs, and with Swagger UI it provides online sandbox for frontend developers.

Swagger is a tool, a specification and a complete framework implementation for producing the visual representation of RESTful Web Services. It enables documentation to be updated at the same pace as the server. When properly defined via Swagger, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Thus Swagger removes the guesswork in calling the service.

**Q: What is Spring Profiles? How do you implement it using Spring Boot?**

Spring Profiles allows users to register beans depending on the profile(dev, test, prod etc). So when the application is running in DEVELOPMENT only certain beans can be loaded and when in PRODUCTION certain other beans can be loaded. Suppose our requirement is that the Swagger documentation be enabled only for the QA environment and disabled for all others. This can be done using Profiles. Spring Boot makes using Profiles very easy.

**Q:What is Spring Batch? How do you implement it using Spring Boot?**

Spring Boot Batch provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. It also provides more advanced technical services and features that will enable extremely high-volume and high performance batch jobs though optimization and partitioning techniques.Simple as well as complex, high-volume batch jobs can leverage the framework in a highly scalable manner to process significant volumes of information.

**Q: How to implement Exception Handling using Spring Boot?**

Spring provides a very useful way to handle exceptions using ControllerAdvice.

We will be implementing a ControlerAdvice class which will handle all exceptions thrown by the controller class.

**Q: What is caching? Have you used any caching framework with Spring Boot?**

A cache is an area of local memory that holds a copy of frequently accessed data that is otherwise expensive to get or compute. Have used Hazelcast for caching.

**Q: What is Apache Kafka? How to integrate it with Spring Boot?**

Apache Kafka is a distributed publish-subscribe messaging system. It is a scalable, fault-tolerant, publish-subscribe messaging system which enables us to build distributed applications. It is an Apache Top Level project. Kafka is suitable for both offline and online message consumption.

**Q: How can we monitor all the Spring Boot Microservices?**

Spring Boot provides actuator endpoints to monitor metrics of individual microservices. These endpoints are very helpful for getting information about applications like if they are up, if their components like database etc are working good. But a major drawback or difficulty about using actuator enpoints is that we have to individually hit the enpoints for applications to know their status or health. Imagine microservices involving 50 applications, the admin will have to hit the actuator endpoints of all 50 applications. To help us deal with this situation, we will be using open source project located at https://github.com/codecentric/spring-boot-admin.

Built on top of Spring Boot Actuator, it provides a web UI to enable us visualize the metrics of multiple applications.

**Q: Have you used any Spring Cloud Components with Spring Boot?**

Have used Spring Cloud components like Netflix Eureka for Service Registration,Ribbon for Load Balancing.

**Q: Can you control logging with Spring Boot? How?**

Yes, we can control logging with Spring Boot by specifying log levels on 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 uses Commons Logging for all internal logging and you can change log levels by adding following lines in the application.properties file:

**logging.level.org.springframework=DEBUG**

**logging.level.com.demo=INFO**

**Q: Can you name some common Spring Boot Starter POMs?**

Some of the most common Spring Boot Start dependencies or POMs are spring-boot-starter, spring-boot-starter-web, spring-boot-starter-test. You can use spring-boot-starter-web to enable Spring MVC in Spring Boot application.

**Q: What are some common Spring Boot annotations?**

Some of the most common Spring Boot annotations are @EnableAutoConfiguration, @SpringBootApplication, @SpringBootConfiguration, and @SpringBootTest.

The **@EnableAutoConfiguration** is used to enable auto-configuration on Spring Boot application, while **@SpringBootApplication** is used on the Main class to allow it to run a JAR file. **@SpringBootTest** is used to run unit test on Spring Boot environment.

**Q: What embedded containers does Spring Boot support?**

Spring Boot support three embedded containers: Tomcat, Jetty, and Undertow. By default, it uses Tomcat as embedded containers but you can change it to Jetty or Undertow.

**Q: What is the difference between an embedded container and a WAR?**

The main difference between an embedded container and a WAR file is that you can Spring Boot application as a JAR from the command prompt without setting up a web server. But to run a WAR file, you need to first set up a web server like Tomcat which has Servlet container and then you need to deploy WAR there.

**Q: Can you change the port of Embedded Tomcat server in Spring boot? If Yes, How?**

Yes, we can change the port of Embedded Tomcat Server in Spring Boot by adding a property called server.port in the application.properties file.

As explained in the previous question, this property file is automatically loaded by Spring Boot and can be used to configure both Spring Boot as well as application code.

**Q: What is Spring Actuator? What are its advantages?**

This is an interesting Spring Boot question and mostly asked on Java roles which also has some support responsibility. Spring Actuator is another cool Spring Boot feature which allows seeing inside a running application.

Yes, you read it correctly. It allows you to see inside an application. Since Spring Boot is all about auto-configuration it makes debugging difficult and at some point in time, you want to know which beans are created in Spring's Application Context and how Controllers are mapped. Spring Actuator provides all that information.

It provides several endpoints e.g. a REST endpoint to retrieve this kind of information over the web. It also provides a lot of insight and metrics about application health e.g. CPU and memory usage, number of threads etc.

It also comes with a remote shell which you can use to securely go inside Spring Boot application and run some command to expose the same set of data. You can even use JMX to control this behavior at runtime.

Btw, it's important to secure your Spring Actuator endpoints because it exposes a lot of confidential information and a potentially dangerous one-two. For example, by using /showdown endpoint you can kill a Spring Boot application.

**Q: What is the difference between @SpringBootApplication and @EnableAutoConfiguration** annotation?

Even though both are essential Spring Boot application and used in the Main class or Bootstrap class there is a subtle difference between them. The @EnableAutoConfiguration is used to enable auto-configuration but @SpringBootApplication does a lot more than that.

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

The @SpringBootApplication is in fact combination of @Configuration, @ComponentScan and @EnableAutoConfiguration annotations. You can also check Spring Boot MasterClass to learn more about this annotation and it's used.

Also, this Spring Boot question was recently asked to one of my friends in his last interview with a big Investment bank. He was interviewing for a front-office Java web application which uses Spring Boot in the back-end.

# **Spring Component Scan :**

When developing Spring Boot applications, you need to tell the Spring Framework where to look for Spring components. Using component scan is one method of asking Spring to detect Spring-managed components. Spring needs the information to locate and register all the Spring components with the application context when the application starts.

Spring can auto scan, detect, and instantiate components from pre-defined project packages. Spring can auto scan all classes annotated with the stereotype annotations @Component, @Controller, @Service, and @Repository

Let's create a simple Spring Boot application to understand how component scanning works in Spring.

We start by writing few components.

import org.springframework.stereotype.Component;

**@Component("demoBeanA")**

public class DemoBeanA {

}

Like wise we have components

**@Component("demoBeanB1")**

**@Component("demoBeanB2")**

**@Component("demoBeanC”)**

**@Component("demoBeanD")**

**public class DemoBeanD {**

**}**

## **The @SpringBootApplication Annotation**

Spring needs to know which packages to scan for annotated components in order to add them to the IoC container. In a Spring Boot project, we typically set the main application class with the @SpringBootApplication annotation. Under the hood, **@SpringBootApplication** is a composition of the @**Configuration, @ComponentScan, and @EnableAutoConfiguration**annotations. With this default setting, Spring Boot will auto scan for components in the current package (containing the @SpringBoot main class) and its sub packages.

The @ComponentScan annotation is used with the @Configuration annotation to tell Spring the packages to scan for annotated components. @ComponentScan also used to specify base packages and base package classes using thebasePackageClasses or basePackages attributes of @ComponentScan.

The basePackageClasses attribute is a type-safe alternative to basePackages. When you specify basePackageClasses, Spring will scan the package (and subpackages) of the classes you specify.

**@Configuration**

**@ComponentScan(basePackages** = {

"guru.springframework.blog.componentscan.example.demopackageA",

"guru.springframework.blog.componentscan.example.demopackageD",

"guru.springframework.blog.componentscan.example.demopackageE"

},

basePackageClasses = DemoBeanB1.class)

public class BlogPostsApplicationWithComponentScan {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationWithComponentScan.class, args);

System.out.println("Contains A " + context.containsBeanDefinition("demoBeanA"));

System.out.println("Contains B2 " + context.containsBeanDefinition("demoBeanB2"));

System.out.println("Contains C " + context.containsBeanDefinition("demoBeanC"));

System.out.println("Contains D " + context.containsBeanDefinition("demoBeanD"));

}

}

The @ComponentScan annotation uses the basePackages attribute to specify three packages (and subpackages) that will be scanned by Spring. The annotation also uses the basePackageClasses attribute to declare the DemoBeanB1 class whose package Spring Boot should scan.

As demoBeanC is in a different package, Spring did not find it during component scanning.

## **Component Scanning Filters**

You can configure component scanning by using different type filters that Spring provides.

Filters can be of two types: include and exclude filters. As their names suggest, include filters specify which types are eligible for component scanning, while exclude filters specify which types are not.

You can use the include and/or exclude filters with or without the default filter. To disable the default filter, set the useDefaultFilters element of the @ComponentScan annotation to false.

@Configuration

@ComponentScan(value = "guru.springframework.blog.componentscan.example.demopackageA",

useDefaultFilters = false)

public class BlogPostsApplicationDisablingDefaultFilters {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationDisablingDefaultFilters.class,args);

System.out.println("Contains A " + context.containsBean("demoBeanA"));

}

}

In the preceding code, the value member defines the specific guru.springframework.blog.componentscan.example.demopackageA package to scan, while the useDefaultFilters member disables the default filter.

**Component Scanning Filter Types**

Spring provides the FilterType enumeration for the type filters that may be used in conjunction with @ComponentScan.

The available FilterType values are:

* **FilterType.ANNOTATION**: Include or exclude those classes with a stereotype annotation
* **FilterType.ASPECTJ**: Include or exclude classes using an AspectJ type pattern expression
* **FilterType.ASSIGNABLE\_TYPE:** Include or exclude classes that extend or implement this class or interface
* **FilterType.REGEX**: Include or exclude classes using a regular expression
* **FilterType.CUSTOM**: Include or exclude classes using a custom implementation of the org.springframework.core.type.TypeFilter interface

### **Include Filters**

With include filters, you can include certain classes to be scanned by Spring. To include assignable type, use the includeFilters element of the @ComponentScan annotation with FilterType.ASSIGNABLE\_TYPE. Using this filter, you can instruct Spring to scan for classes that extends or implements the class or interface you specify.

@Configuration

**@ComponentScan(basePackages = {"guru.springframework.blog.componentscan.example.demopackageA",**

**"guru.springframework.blog.componentscan.example.demopackageB"},**

**includeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE, value = DemoBeanB2.class),**

**useDefaultFilters = false)**

public class BlogPostsApplicationIncludeFilter {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationIncludeFilter.class,args);

System.out.println("Contains A " + context.containsBean("demoBeanA"));

System.out.println("Contains B1 " + context.containsBean("demoBeanB1"));

System.out.println("Contains B2 " + context.containsBean("demoBeanB2"));

System.out.println("Contains B3 " + context.containsBean("demoBeanB3"));

}

}

**Exclude Filters :** The @ComponentScan annotation enables you to exclude those classes that you do not want to scan.

@Configuration

@ComponentScan(basePackageClasses = {DemoBeanB1.class},

excludeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE,

value = DemoBeanB2.class))

In this code, the nested annotation @ComponentScan.Filter is used to specify the filter type as FilterType.ASSIGNABLE\_TYPE and the base class that should be excluded from scanning.

**Bean Overriding :** Spring beans are identified by their names within an ApplicationContext.

Thus, bean overriding is a default behavior that happens when we define a bean within an ApplicationContext which has the same name as another bean. It works by simply replacing the former bean in case of a name conflict.

Starting in Spring 5.1, the **[BeanDefinitionOverrideException](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/support/BeanDefinitionOverrideException.html)**was introduced to allow developers to automatically throw the exception to prevent any unexpected bean overriding. By default, the original behavior is still available which allows bean overriding.

Spring's @Bean annotation is a very common way of defining a bean.

Thus, another option is to set the name property of @Bean annotation:

@Bean("testBean1")

public TestBean1 testBean() {

return new TestBean1();

}

@Bean("testBean2")

public TestBean1 testBean() {

return new TestBean2();

}

Another way to define a bean is with stereotype annotations. With Spring's @ComponentScan feature enabled, we can define our bean names at the class level using the @Component annotation:

@Component("testBean1")

class TestBean1 {

private String name; // standard getters and setters

}

@Component("testBean2")

class TestBean2 {

private String name; // standard getters and setters

}

**To enable bean overriding, let's set the** spring.main.allow-bean-definition-overriding property to true in our application.properties file:

**spring.main.allow-bean-definition-overriding=true**

By doing this, we are telling Spring Boot to allow bean overriding without any change to bean definitions.