Spring Boot

What is Spring Boot ?

Spring Boot is an open-source java-based framework that helps in bootstrapping the Spring Application. “Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can just run“.

Core Principles of Spring Boot?

four core principles: autoconfiguration, opinionated, convention over configuration, and eliminates boilerplate code.

Why is Spring Boot so popular?

1. Spring Boot speeds up the project development.

2. It provides an easier and faster way to configure, set up & run both web-based as well as simple applications.

3. No web servers like (Tomcat, Jetty, Undertow) are required to configure separately. Therefore, no need to deploy any war file manually.

4. Reduces XML configurations and promotes annotations.

5. Combines more than one annotations of Spring framework & replaces them by introducing a single annotation. For example, @SpringBootApplication is a combination of three annotations : @EnableAutoConfiguration, @Configuration, @ComponentScan

6. Reduces the efforts of developer to deliver an enterprise-level application.

7. Provides ready-made functionalities that are the most common and applicable to any project. Hence, it eliminates boiler plate code.

8. Creates stand-alone application that can be run using jar file.

9. Automatically configures Spring and third party libraries whenever possible.

10. Provides production-ready features such as health checks, metrics and externalized configurations.

11. Provides easy integration mechanism with many third party tools.

12. Spring Boot Application easily integrates with its ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security, Spring Web MVC etc.

13. It integrates with databases very easily. Sometimes, no need to provide database specific properties like driver, dialects etc., as it automatically picks these property values from database URL.

14. It provides developers ready-made starter projects to quickly work on functionalities rather than wasting time in setting up initial configurations.

15. Spring Boot also provides command line interface(CLI) to develop and test applications quickly.

16. It provides support of most commonly used build tools like Maven & Gradle.

How Does Spring Boot Work?

Spring Boot works by offering a set of starter dependencies that we can use to quickly bootstrap a Spring-based application. These starter dependencies incorporate all the required libraries and configurations necessary to get an application up and running. Developers can use these starter dependencies to create a new project and get started quickly.

What are the advantages of Spring Boot?

1) Spring Boot Starters
2) Reducing Boilerplate Code
3) Embedded Servers
4) Easier to connect with Databases
5) Supports development of Microservices
6) Default setup for Testing
7) Run Application as a jar file
8) Spring Boot Profile to easy switch environment
9) Spring Boot Actuator for Application Monitoring

10) Increases Productivity and minimises Development time

How many ways are there to create a Spring Boot Application?

1) Using Spring Boot CLI Tool
2) Using Spring STS IDE
3) Using Spring Initializr Website

What are the Starters in Spring Boot ?

In Spring Boot, Starters are the ready made small projects that we can include in our application. Officially they are called dependency descriptors. The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies. For example, if you want to get started with Spring Web Application, include the *spring-boot-starter-web* dependency in your project. However, if you are using STS as an IDE, you need to just search for 'Web' and select 'Spring Web'. The STS will automatically add the 'spring-boot-starter-web' dependency in your pom.xml.

Spring Boot Features

CommandLineRunner

If we want to run some specific code once the Spring Boot Application has started, we can implement the CommandLineRunner interface. This interface provides a single run() method which is called just before SpringApplication.run() completes. Similarly, we have one more interface ApplicationRunner that we can also use. For example, the following code demonstrates the use of CommandLineRunner with the run() method.

```
@Component
public class MyCommandLineRunner implements CommandLineRunner {

    @Override
    public void run(String... args) {
       // code that you want to run only once
    }
}
```

If there are several CommandLineRunner or ApplicationRunner beans defined that must be called in a specific order, you can additionally use the @Order annotation or implement the org.springframework.core.Ordered interface.

Externalized Configuration

Spring Boot Application Properties Files

In a Spring Boot Application, 'application.properties' is an input file to set the application up. Unlike the standard Spring framework configuration, this file is auto detected in a Spring Boot Application. It is placed inside the "src/main/resources" directory. Therefore, we don't need to specially register a PropertySource, or even provide a path to a property file.

An application.properties file stores various properties in key=value format. These properties are used to provide input to Spring container objects, which behaves like one time input or static data.

What types of keys does an application.properties contain?

1. Pre-defined Keys
2. Programmer-defined Keys

How to load Properties files

Spring @PropertySource annotation is used to provide properties file to Spring Environment. This annotation is used with @Configuration classes. Spring PropertySource annotation is repeatable, means you can have multiple PropertySource on a Configuration class. This feature is available if you are using Java 8 or higher version.

YAML files cannot be loaded by using the @PropertySource

```
@PropertySources({
    @PropertySource("classpath:foo.properties"),
    @PropertySource("classpath:bar.properties")
})
public class PropertiesWithJavaConfig {
```

```
    //...
}
```

To read one key from the properties file, we need to declare one variable in our class. Moreover, we need to apply the @Value annotation at the variable with the key as a parameter to it. It's like @Value("${key}")

```
@Value( "${jdbc.url}" )
private String jdbcUrl;
```

@Value for default Value

Suppose we have not defined a property in the properties file. In that case we can provide a default value for that property. Here is the example:

```
@Value("${emp.department:Admin}")
private String empDepartment;
```

How does the @Value work internally in Spring?

First 'application.properties' file is loaded using code like @PropertySource("classpath:application.properties").

Now, Spring container creates one MEMORY "Environment" (key-val pairs) that holds all the key-values given by Properties file.

We can read those values into variable using Syntax @Value("${key}").

Then, @Value will search for key in Environment memory. If found then value is updated into container object.

How to map multiple properties with a single variable? When to use @ConfigurationProperties annotation?

We have a requirement where we need to read multiple properties from properties file in our code. Here, we will make use of @ConfigurationProperties annotation. Furthermore, @ConfigurationProperties also supports working with Collections and Bean Type (class Type).

What are the general guidelines/rules to use @ConfigurationProperties annotation?

1) The prefix provided inside the @ConfigurationProperties annotation must match with the prefix given in the properties file, else variable will not get the correct values.

2) We must generate getters and setters for the variables that we used to map with the keys of properties file.

♥ Note: Since @Value uses refection based data read, hence getters & setters are not required in that case.

```
product.app.id=1024

product.app.code=QS5329D

product.app.version=3.34
```

```
@ConfigurationProperties("product.app")
```

Now, create a bean class with getter and setter methods and annotate it with @ConfigurationProperties.

What is YAML(.yml) file?

YAML is a latest format to define key:val pairs without having duplicate words in more than one key. In other words, YAML is a convenient format for specifying hierarchical configuration data. It uses "Snake YAML" API and it is added by default in every Spring Boot Application.

Unlike .properties file, dot(.) and equals(=) are not allowed, but only colon(:) is allowed.

If you have configuration files with both '.properties' and '.yml' format in the same location, '.properties' takes precedence.

Can we modify/rename file name application.properties in Spring Boot?

Yes we can, but it is not loaded by Spring Boot by default. By default, Spring Boot checks 'application.properties' under location 'src/main/resources'. If we want to load other files, follow below steps:

1) Create your custom file at the same location ('src/main/resources')
2) Apply @PropertySource annotation at starter/runner class and provide the name of your custom file as a parameter to it.
For example: @PropertySource("classpath:xyz.properties")
here, the classpath is 'src/main/resource' folder

Alternatively, we can also keep this file inside project folder (file:/ = project folder)

If a key is present in both application.properties and our custom properties file, then which one will be selected?

Key from 'application.properties' will always have a higher priority.

What is Spring Profile?

Profiles are the group of configuration properties. These configuration properties belong to a specific environment. If all the configuration properties belong to development environment, we can call it development profile. Similarly, based on the specific configuration properties it holds, we can call other profiles as test profile, prod profile etc. Spring profiles provide a way to separate out parts of our application configuration and make it be available only in specific environments.

Programmatically, a profile is a property file with extension '.properties' similar to 'application.properties' file. In spring Boot, apart from '.properties', a profile can also have '.yml' extension.

Usages of Profiles

Configuration Properties: Profile controls which configuration properties should be active.
Beans: Profile controls which beans should be loaded into the Spring Container. It offers a provision to register beans by condition.

What is the Default Profile?

Spring Boot comes with a property file, named 'application.properties' file by default. Therefore, this file is the default profile. Similarly, 'application.yml' file will also be the default profile. The default profile is always active. Spring Boot loads all properties from the default profile first.

we should keep all properties in default profile (application. properties) which are common across all profiles.

How to create a Spring Profile?

Spring Boot comes with a property file, named 'application.properties' file by default.

Please note the naming convention:
application-<environment>.properties

1) application-dev.properties

2) application-test.properties

3) application-prod.properties

How to Activate a Particular Profile?

By setting 'spring.profile.active' in application.properties

By setting in JVM System Parameter -Dspring.profiles.active=dev

By implementing WebApplicationInitializer Interface

By setting active profiles in pom.xml

Priority on Using Multiple Approaches

Suppose we have used multiple approaches to set active profile in the same project, then Spring will prioritize approaches in below order.

1) By Setting a context parameter in web.xml

2) By Implementing WebApplicationInitializer Interface

3) By Setting JVM System Parameter

4) By Setting active profiles on pom.xml

How To Use @Profile Annotation?

By using @Profile annotation, we can make a bean belong to a particular profile. The @Profile annotation simply takes the names of one or multiple profiles. In simple words, @Profile annotation associates a bean to a particular profile. For example, let's consider a scenario: We need a bean that should only be active during development, but not production. It will only be present in the container during development. In production, it won't be active. Below code demonstrates the concept:

```
@Profile("dev")

@Component

public class MyBeanConfig{...}
```

Furthermore, profile names can also be prefixed with a NOT operator, e.g., !dev, to exclude them from a profile. For example, the below component is activated only if dev profile is not active:

```
@Profile("!dev")

@Component

public class MyBeanConfig {...}
```

Furthermore, We can also activate multiple profiles at a time by using @Profile annotation as shown below:

How to check which Profile is currently Active?

1. Using Environment Object
2. By injecting the property 'spring.profiles.active

Component Scanning in a Spring Boot Application

If your package doesn't come under the hierarchy of the package containing the Main class, then there is a need for explicit component scanning.

@Component

Indicates that the class is a Spring-managed bean/component. *@Component* is a class level annotation. Applying *@Component* annotation on a class means that we are marking the class to work as Spring-managed bean/component.

During the component scanning, Spring Framework automatically discovers the classes annotated with *@Component*, It registers them into the Application Context as a Spring Bean.

```
@Component
class MyBean { }
```

Spring will create a bean instance with name ‘myBean’.

```
@Component("myTestBean")
class MyBean { }
```

@Controller

@Controller tells Spring Framework that the class annotated with *@Controller* will work as a controller in the Spring MVC project.

@RestController

@RestController tells Spring Framework that the class annotated with *@RestController* will work as a controller in a Spring REST project.

@Service

@Service tells Spring Framework that the class annotated with *@Service* is a part of service layer and it will include business logics of the application.

@Repository

@Repository tells Spring Framework that the class annotated with @Repos*itory* is a part of data access layer and it will include logics of accessing data from the database in the application.

Apart from Stereotype annotations, we have two annotations that are generally used together: @Configuration & @Bean.

@Configuration

We apply this annotation on classes. When we apply this to a class, that class will act as a configuration by itself. Generally the class annotated with *@Configuration* has bean definitions as an alternative to <bean/> tag of an XML configuration. It also represents a configuration using Java class.

@Bean

We use *@Bean* at method level. It creates Spring beans and generally used with *@Configuration*. As aforementioned, a class with *@Configuration (we* can call it as a Configuration class) will have methods to instantiate objects and configure dependencies. Such methods will have *@Bean* annotation.

```
@Configuration
public class AppConfig {
      @Bean
      public Employee employee() {
          return new Employee();
      }
     @Bean
     public Address address() {
         return new Address();
      }
}
```

@Value for multiple values

Sometimes, we need to inject multiple values of a single property. We can conveniently define them as comma-separated values for the single property in the properties file. Further, we can easily inject them into property that is in the form of an array. For example:

```
@Value("${columnNames}")
private String[] columnNames;
```

Spring Boot Specific Annotations

@SpringBootApplication (@Configuration + @ComponentScan + @EnableAutoConfiguration)

@EnableAutoConfiguration: enables the auto-configuration feature of Spring Boot.
@ComponentScan: enables @Component scan on the package to discover and register components as beans in Spring's application Context.
@Configuration: allows to register extra beans in the context or imports additional configuration classes.

@EnableAutoConfiguration

@EnableAutoConfiguration enables auto-configuration of beans present in the classpath in Spring Boot applications. In a nutshell, this annotation enables Spring Boot to auto-configure the application context. Therefore, it automatically creates and registers beans that are part of the included jar file in the classpath and also the beans defined by us in the application. For example, while creating a Spring Boot starter project when we select Spring Web and Spring Security dependency in our classpath, Spring Boot auto-configures Tomcat, Spring MVC and Spring Security for us.

Moreover, Spring Boot considers the package of the class declaring the *@EnableAutoConfiguration* as the default package. Therefore, if we apply this annotation in the root package of the application, every sub-packages & classes will be scanned. As a result, we won't need to explicitly declare the package names using *@ComponentScan*.

Furthermore, *@EnableAutoConfiguration* provides us two attributes to manually exclude classes from auto-configurations. If we don't want some classes to be auto-configured, we can use the exclude attribute to disable them. Another attribute is excludeName to declare a fully qualified list of classes to exclude. For example, below are the codes.

Use of 'exclude' in *@EnableAutoConfiguration*

```
@Configuration

@EnableAutoConfiguration(exclude={WebSocketMessagingAutoConfigura
tion.class})

public class MyWebSocketApplication {

        public static void main(String[] args) {

            ...

        }

}
```

Use of ‘excludeName’ in *@EnableAutoConfiguration*

```
@Configuration

@EnableAutoConfiguration(excludeName =
{"org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMes
sagingAutoConfiguration"})

public class MyWebSocketApplication {

      public static void main(String[] args) {

         ...

      }

}
```

@SpringBootConfiguration

@ConfigurationProperties

Spring Framework provides various ways to inject values from the properties file. One of them is by using *@Value* annotation. Another one is by using *@ConfigurationProperties* on a configuration bean to inject properties values to a bean. But what is the difference among both ways and what are the benefits of using *@ConfigurationProperties*, you will understand it at the end. Now Let's see how to use *@ConfigurationProperties* annotation to inject properties values from the application.properties or any other properties file of your own choice.

First, let's define some properties in our application.properties file as follows. Let's assume that we are defining some properties of our development working environment. Therefore, representing properties name with prefix 'dev'.

```
dev.name=Development Application

dev.port=8090

dev.dburl=mongodb://mongodb.example.com:27017/

dev.dbname=employeeDB

dev.dbuser=admin

dev.dbpassword=admin
```

Now, create a bean class with getter and setter methods and annotate it with @ConfigurationProperties.

```
@ConfigurationProperties(prefix="dev")

public class MyDevAppProperties {

    private String name;

    private int port;

    private String dburl;

    private String dbname;

    private String dbuser;

    private String dbpassword;



   //getter and setter methods

}
```

Here, Spring will automatically bind any property defined in our property file that has the prefix '*dev*' and the same name as one of the fields in the MyDevAppProperties class.

Next, register the *@ConfigurationProperties* bean in your *@Configuration* class using the *@EnableConfigurationProperties* annotation.

```
@Configuration

@EnableConfigurationProperties(MyDevAppProperties.class)

public class MySpringBootDevApp { }
```

Finally create a Test Runner to test the values of properties as below.

```
@Component

public class DevPropertiesTest implements CommandLineRunner {

    @Autowired

    private MyDevAppProperties devProperties;

    @Override

    public void run(String... args) throws Exception {

        System.out.println("App Name = " + devProperties.getName());

        System.out.println("DB Url = " + devProperties.getDburl());

        System.out.println("DB User = " + devProperties.getDbuser());

    }

}
```

We can also use the *@ConfigurationProperties* annotation on *@Bean*-annotated methods.

@EnableConfigurationProperties

In order to use a configuration class in our project, we need to register it as a regular Spring bean. In this situation *@EnableConfigurationProperties* annotation support us. We use this annotation to register our configuration bean (a *@ConfigurationProperties annotated class) in a Spring context*. This is a convenient way to quickly register *@ConfigurationProperties* annotated beans. Moreover, It is strictly coupled with *@ConfiguratonProperties.* For

example, you can refer *@ConfigurationProperties* from the previous section.

@EnableConfigurationPropertiesScan

@EnableConfigurationPropertiesScan annotation scans the packages based on the parameter value passed into it and discovers all classes annotated with *@ConfiguratonProperties* under the package. For example, observe the below code:

```
@SpringBootApplication

@EnableConfigurationPropertiesScan("com.dev.spring.test.annotation")

public class MyApplication { }
```

From the above example, *@EnableConfigurationPropertiesScan* will scan all the *@ConfiguratonProperties* annotated classes under the package "com.dev.spring.test.annotation" and register them accordingly.

@EntityScan and @EnableJpaRepositories

Spring Boot annotations like *@ComponentScan*, *@ConfigurationPropertiesScan* and even *@SpringBootApplication* use packages to define scanning locations. Similarly *@EnityScan* and *@EnableJpaRepositories* also use packages to define scanning locations. Here, we use *@EntityScan* for discovering entity classes, whereas *@EnableJpaRepositories* for JPA repository classes by convention. These annotations are generally used when your discoverable classes are not under the root package or its sub-packages of your main application.

Remember that, *@EnableAutoConfiguration(*as part of *@SpringBootApplication)* scans all the classes under the root package or its sub-packages of your main application. Therefore, If the repository classes or other entity classes are not placed under the main application package or its sub package, then the relevant package(s) should be declared in the main application configuration class with

@EntityScan and *@EnableJpaRepositories* annotation accordingly. For example, observe the below code:

```
@EntityScan(basePackages = "com.dev.springboot.examples.entity")


@EnableJpaRepositories(basePackages =
"com.dev.springboot.examples.jpa.repositories")
```

@Lazy

By default, Spring creates all singleton beans eagerly at the startup/bootstrapping of the application context. However, in some cases when we need to create a bean, not at the application context startup, but when we request it intentionally. In that case we apply *@Lazy*. When we put *@Lazy* annotation over the *@Configuration* class, it indicates that all the methods with *@Bean* annotation should be loaded lazily.

@Profile

We use the *@Profile* annotation which indicates that we are mapping the bean to that particular profile. It indicates that beans will be only initialised if the defined profiles are active.

@Scope

The scope of a bean indicates the life cycle and visibility of that bean in the contexts in which it is used. Spring framework defines 6 types of scopes as per the latest version.

singleton
prototype
request
session
application
Websocket

@DependsOn

When one bean has dependency on other bean we use *@DependsOn* annotation. Also, if you need to initialize any bean before another bean, *@DependsOn* will help you to do this job. While creating bean we need to define value of *@DependsOn* attribute as a dependent bean. Moreover, the *@DependsOn* attribute can explicitly force one or more beans to be initialized before the current bean is initialized.

```
@Component
public class BeanA { }

@Component
public class BeanB { }

@Component
@DependsOn(value = {"beanA","beanB"})
public class BeanC { }
```

@Order

The *@Order* annotation defines the sorting order of an annotated component or a bean.
It has an optional 'value' argument which determines the execution order of the component.

♥ Note : Execution order will be as below:

First components with order value of negative number

Then components with order value of positive number

Then no order value components in alphabetical order of their names

@Primary

When we have multiple beans of the same type, we use *@Primary* to give higher preference to a particular bean. For example, Let's assume that we have two beans PermanentEmployee and ContractEmployee of type Employee. Here, we want to give preference to PermanentEmployee. Let's observe below code to know how we will apply *@Primary* to give the preference.

```
@Component
public class ContractEmployee implements Employee { }

@Component
@Primary
public class PermanentEmployee implements Employee { }

@Service
public class EmployeeService {
    @Autowired
    private Employee employee;
    public Employee getEmployee() {
        return employee;
    }
}
```

@Conditional

@Conditional indicates that a component is only eligible for registration when all specified conditions match.
If a *@Configuration* class is marked with *@Conditional*, all of the *@Bean* methods, *@Import* annotations, and *@ComponentScan* annotations associated with that class will be subject to the conditions.

```
@Bean
@ConditionalOnJava(value = JavaVersion.NINE)
public JavaBean getJavaBean(){
    return new JavaBean();
}
```

Annotations on Bean Injection

@Autowired, @Resource, @Inject

These annotations belong to dependency injection.
Since *@Autowired* is part of the Spring framework directly, we as a developer always prefer to use this in our project. In this case Dependency Injection is handled completely by the Spring Framework. Although Each of these annotations can resolve dependencies either by field injection, constructor injection or by setter injection.

@Autowired

@Autowired facilitates to resolve and inject the object dependency implicitly. It internally uses setter or constructor injection. We can't use *@Autowired* to inject primitive and string values. It works with reference only.

@Autowired Execution Precedence

This annotation has execution paths as below listed by precedence:
First Match by Type
Then Match by Qualifier
Then Match by Name

@Qualifier

The *@Qualifier* annotation provides additional information to *@Autowired* annotation while resolving bean dependency. If more than one bean of the same type is available in the container, we need to tell container explicitly which bean we want to inject, otherwise the framework will throw an NoUniqueBeanDefinitionException, indicating that more than one bean is available for autowiring. Here, *@Qualifier* does the required job.

@Primary vs @Qualifier

We can also use *@Primary* annotation when we need to decide which bean to inject in case of ambiguity. Similar to *@Qualifier*, this annotation also defines a preference when multiple beans of the same type are present. In that situation, the bean annotated with the *@Primary* will be used unless otherwise indicated. However, this annotation is useful when we want to specify which bean of a certain type should be injected by default. Mainly, we use *@Primary* in the sense of a default, while *@Qualifier* in the sense of a very specific. In case If both the *@Qualifier* and *@Primary* annotations are present, then the *@Qualifier* annotation will take precedence.

Annotations on Bean State

In Spring, we can either implement InitializingBean and DisposableBean interfaces or specify the init-method and destroy-method in bean configuration file for the initialization and destruction callback function. Here, we will discuss about two annotations *@PostConstruct* and *@PreDestroy* to do the same thing.

♦ Note: Please note that the *@PostConstruct* and *@PreDestroy* annotations do not belong to the Spring. They are part of JEE library and exits in common-annotations.jar. Since Java EE has deprecated in Java 9 and also removed in Java 11 we have to add an additional dependency to use these annotations: javax.annotation-api. If you are using Java 8 or older version, there is no need to include any additional dependency.

@PostConstruct

Spring calls methods annotated with *@PostConstruct* only once, just after the initialization of bean properties. Annotated method is executed after dependency injection is done to perform initialization. The method annotated with *@PostConstruct* can have any access level except static.

@PreDestroy

Spring calls methods method annotated with *@PreDestroy* only once, just before Spring removes our bean from the application context. Annotated method is executed before the bean is

destroyed, e.g. on the shutdown. Similar to *@PostConstruct*, the method annotated with *@PostConstruct* can have any access level except static.

```
public class MyTestClass {
    @PostConstruct
    public void postConstruct() throws Exception {
        System.out.println("Init method after properties are set");
    }
    @PreDestroy
    public void preDestroy() throws Exception {
      System.out.println("Spring Container is destroyed! clean up");
    }
}
```

What is the difference between @Primary and @Qualifier annotations?

1) @Primary resolves the ambiguity problem by making one of the dependent Spring Bean as the primary Spring Bean to inject, whereas @Qualifier resolves the same problem by taking that dependent Spring Bean whose bean Id is specified in it.

2) @Primary makes IOC container to inject its Spring Bean as dependent to target bean using 'By Type' mode of auto-wiring, whereas @Qualifier does the same thing using 'By Name' mode of auto-wiring.

3) @Primary is applicable at the class level, whereas @Qualifier is applicable at field level, parameter level, and method level.