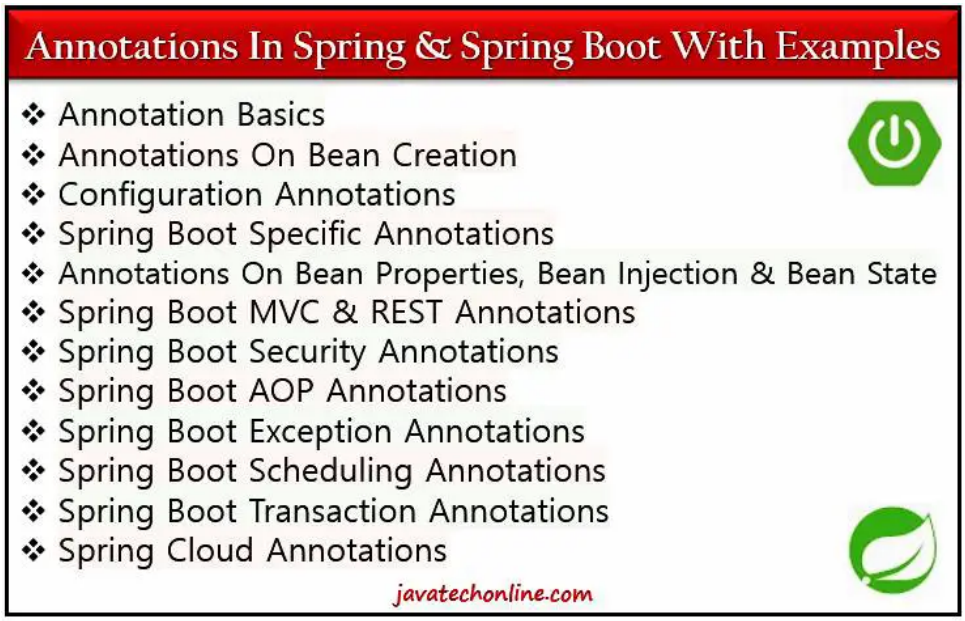
Making Java easy to learn

Java Technology and Beyond

Spring Boot Annotations With Examples



Prior to Annotations, the Spring Framework's configuration was largely dependent on XMLs. Using XML configurations was not only a tedious process, but also an error-prone. If you committed any syntactical mistake in XML, sometimes it takes time to fix. But now-a-days annotations, particularly Spring Boot Annotations provide us remarkable capabilities in configuring Spring Framework's behavior. Moreover, Annotations caused major changes in programming style and slowly making the XML-based configurations outdated. The Java Programming introduced support for Annotations from JDK 1.5. However Spring Framework started supporting annotations from the release 2.5. Obviously, we are going to discuss about Spring Boot Annotations With Examples and their usages.

Here in this article on 'Spring Boot Annotations With Examples', we will discuss about all annotations that we use in a Spring Boot Application. Annotations which are not part of this article, will be included in other respective articles. Link will be provided here only for easy navigation. In addition, you can also check one more article 'Annotations in Java' in order to know annotation basics and how to cre

TOP

custom annotations in Java. Let's start discussing our topic 'Spring Boot Annotations With Examples'.

Table of Contents

TOP

Annotation Basics

Before discussing about 'Spring Boot Annotations With Examples', let's first talk about some basic terminologies used during the explanation of annotations.

What is The IoC container?

In a nutshell, the IoC is a container that injects dependencies while creating the bean. IoC stands for 'Inversion Of Control'. Instead of creating objects by the developer, the bean itself controls the instantiation or association of its dependencies by using direct construction of classes with the help of the IoC container. Hence, this process is known as 'Inversion of control'. Sometimes we also call it Spring Container in short.

The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework's IoC container. ApplicationContext is a sub-interface of BeanFactory.

What is an Application Context in Spring Framework?

When you create a project in Spring or Spring Boot, a container or wrapper gets created to manage your beans. This is nothing but Application Context. However Spring supports two containers : Bean Factory and Application Context. In short, the BeanFactory provides the configuration framework and basic functionality. On the other hand, ApplicationContext adds more enterprise-specific functionality including easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications. The ApplicationContext is a complete superset of the BeanFactory and is used exclusively in this topic in descriptions of Spring's IoC container. Spring Framework recommends to use Application Context to get the full features of the framework. Moreover, Dependency injection and auto-wiring of beans is done in Application Context.

The interface org.springframework.context.ApplicationContext represents the Spring IoC container and it manages the process of instantiating, configuring, and assembling the beans. The container g
instructions on what objects to instantiate, configure, and assemble by reading configuration

TOP

metadata. The configuration metadata can be represented in three formats: XML, Java annotations, or Java code. Since this is out of our topic 'Spring Boot Annotations With Examples', we have discussed about this in detail in a separate article 'Spring Core Tutorial'.

What is Spring Bean or Components?

During Application startup, Spring instantiates objects and adds them to the Application Context(IoC container). These objects in the Application Context are called 'Spring Beans' or 'Spring Components'. As they are managed by Spring, therefore we also call them Spring-managed Bean or Spring-managed Component.

What is Component Scanning?

The process of discovering classes that can contribute to the Application Context, is called Component Scanning. During Component Scanning, if Spring finds a class annotated with a particular annotation, It will consider this class as a candidate for Spring Bean/Component and adds it to the Application Context. Spring explicitly provides a way to identify Spring bean candidates via the *@ComponentScan* annotation.

When to use Component Scanning in a Spring Boot Application?

By default, the *@ComponentScan* annotation will scan for components in the current package and its all sub-packages. If it is a Spring Boot application, then all the packages under the package containing the Main class (a class annotated with *@SpringBootApplication*) will be covered by an implicit component scan. So 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.

Spring Annotations vs Spring Boot Annotations

As we know that Spring Boot Framework is created using libraries of the Spring Framework and removed the XML configuration approach. However, all the Spring Framework's annotations are still applicable for Spring Boot. Furthermore, Spring Boot has provided some additional annotations which are specific to Spring Boot only. In some cases, Spring Boot created annotations after adding more than one annotations of the Spring Framework. Here, we will learn most commonly used annotations under our topic of discussion 'Spring Boot Annotations With Examples', whether it is a part of Spring Framework or Spring Boot.

Annotations that Supports to create a Spring Bean

TOP

Let's start discussing our topic 'Spring Boot Annotations with Examples' with the annotations to create a Bean. Needless to say, we can't work on Spring Boot/Spring framework without creating a Bean. Now you can imagine how important is that!

Next annotations in our topic 'Spring Boot Annotations With Examples' are the stereotype annotations.

@Component

@Complonent annotation is also an important & most widely used at the class level. This is a generic stereotype annotation which indicates that the class is a Spring-managed bean/component. *@Component* is a class level annotation. Other stereotypes are a specialization of *@Component*. During the component scanning, Spring Framework automatically discovers the classes annotated with *@Componen*t, It registers them into the Application Context as a Spring Bean. Applying *@Component* annotation on a class means that we are marking the class to work as Spring-managed bean/component. For example, observe the code below:

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

On writing a class like above, Spring will create a bean instance with name 'myBean'. Please keep in mind that, By default, the bean instances of this class have the same name as the class name with a lowercase initial. However, we can explicitly specify a different name using the optional argument of this annotation like below.

```
@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

TOP

@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. Moreover the class will have methods to instantiate and configure the dependencies. For example :

```
@Configuration
public class AppConfig {
   @Bean
   public RestTemplate getRestTemplate() {
     RestTemplate restTemplate = new RestTemplate();
     return restTemplate();
   }
}
```

♥ The benefit of creating an object via this method is that you will have only one instance of it. You don't need to create the object multiple times when required. Now you can call it anywhere in your code.

@Bean

This is the basic annotation and important for our topic 'Spring Boot Annotations With Examples'.

We use *@Bean* at method level. If you remember the xml configuration of a Spring, It is a direct analog of the XML <bean/> element. 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.

By default, the bean name will be the same as the method name. It instantiates and returns the

TOP

bean. The annotated method produces a bean managed by the Spring IoC container. For example:

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

For the sake of comparison, the configuration above is exactly equivalent to the following Spring XML:

```
<beans>
    <bean name="employee" class="com.dev.Employee"/>
    <bean name="address" class="com.dev.Address"/>
</beans>
```

The annotation supports most of the attributes offered by <bean/>, such as: init-method, destroy-method, autowiring, lazy-init, dependency-check, depends-on and scope.

@Bean vs @Component

@Component is a class level annotation whereas ***@Bean*** is a method level annotation and name of the method serves as the **bean** name. **@Bean** annotation has to be used within the class and that class should be annotated with ***@Configuration.*** However, ***@Component*** needs not to be used with the *@Configuration*. ***@Component*** auto detects and configures the beans using classpath scanning, whereas @Bean explicitly declares a single bean, rather than letting Spring do it automatically.

Configuration Annotations

Next annotations in our article 'Spring Boot Annotations with Examples' are for Configurations. Since Spring Framework is healthy in configurations, we can't avoid learning annotations on configurations. No doubt, they save us from complex coding effort.

TOP

@ComponentScan

Spring container detects Spring managed components with the help of *@ComponentScan*. Once you use this annotation, you tell the Spring container where to look for Spring components. When a Spring application starts, Spring container needs the information to locate and register all the Spring components with the application context. However It can auto scan all classes annotated with the stereotype annotations such as *@Component, @Controller, @Service*, and *@Repository* from pre-defined project packages.

The *@ComponentScan* annotation is used with the *@Configuration* annotation to ask Spring the packages to scan for annotated components. *@ComponentScan* is also used to specify base packages and base package classes using basePackages or basePackageClasses attributes of *@ComponentScan*. For example :

```
import com.springframework.javatechonline.example.package2.Bean1;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@Configuration
@ComponentScan(basePackages = {"com.springframework.javatechonline.example.package1",
                               "com.springframework.javatechonline.example.package3",
                               "com.springframework.javatechonline.example.package4"},
                basePackageClasses = Bean1.class
                )
public class SpringApplicationComponentScanExample {
        ....
}
```

Here the *@ComponentScan* annotation uses the basePackages attribute to specify three packages including their subpackages that will be scanned by the Spring container. Moreover the annotation also uses the basePackageClasses attribute to declare the Bean1 class, whose package Spring Boot will scan.

Moreover, In a Spring Boot project, we typically apply the @SpringBootApplication annotation on the main application class. Internally, *@SpringBootApplication* is a combination of the *@Configuration, @ComponentScan,* and *@EnableAutoConfiguration* annotations. Further, with this default setting, Spring Boot will auto scan for components in the current package (containing the SpringBoot main class) and its sub packages.

@Import

Suppose we have multiple Java configuration classes annotated by *@Configuration*. *@Import* ir

TOP

one or more Java configuration classes. Moreover, it has the capability to group multiple configuration classes. We use this annotation where one *@Configuration* class logically imports the bean definitions defined by another. For example:

```
@Configuration
@Import({ DataSourceConfig.class, MyCustomConfig.class })
public class AppConfig extends ConfigurationSupport {
    // @Bean methods here that can reference @Bean methods in DataSourceConfig or MyCustomConfig
}

@Configuration
public class DataSourceConfig  {...}

@Component
public class MyCustomConfig {...}
```

@PropertySource

If you create a Spring Boot Starter project using an IDE such as STS, application.properties comes under resources folder by default. In contrast, You can provide the name & location of the properties file (containing the key/value pair) as per your convenience by using *@PropertySource*. Moreover, this annotation provides a convenient and declarative mechanism for adding a PropertySource to Spring's Environment. For example,

```
@Configuration
@PropertySource("classpath:/com/dev/javatechonline/app.properties")
public class MyClass {
}
```

@PropertySources (For Multiple Property Locations)

Of course, If we have multiple property locations in our project, we can also use the *@PropertySources* annotation and specify an array of *@PropertySource*.

```
@Configuration
@PropertySources({
        @PropertySource("classpath:/com/dev/javatechonline/app1.properties"),
        @PropertySource("classpath:/com/dev/javatechonline/app2.properties")
```

TOP

```
})
public class MyClass { }
```

However, we can also write the same code in another way as below. The *@PropertySource* annotation is repeatable according to Java 8 conventions. Therefore, if we're using Java 8 or higher, we can use this annotation to define multiple property locations. For example:

```
@Configuration
@PropertySource("classpath:/com/dev/javatechonline/app1.properties")
@PropertySource("classpath:/com/dev/javatechonline/app2.properties")
public class MyClass { }
```

♥ **Note :** If there is any conflict in names such as the same name of the properties, the last source read will always take precedence.

@Value

We can use this annotation to inject values into fields of Spring managed beans. We can apply it at field or constructor or method parameter level. For example, let's first define a property file, then inject values of properties using *@Value*.

```
server.port=9898
server.ip= 10.10.10.9
emp.department= HR
columnNames=EmpName,EmpSal,EmpId,EmpDept
```

Now inject the value of server.ip using *@Value* as below:

```
@Value("${server.ip}")
private String serverIP;
```

@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:

TOP

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

Here, the value *Admin* will be injected for the property *emp.department*. However, if we have defined the property in the properties file, the value of property file will override it.

♥ **Note :** If the same property is defined as a system property and also in the properties file, then the system property would take preference.

@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

Now it's time to extend our topic 'Spring Boot Annotations with Examples' with Spring Boot Specific Annotations. They are discovered by Spring Boot Framework itself. However, most of them internally use annotations provided by Spring Framework and extend them further.

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

Everyone who worked on Spring Boot must have used this annotation either intentionally or unintentionally. When we create a Spring Boot Starter project using an IDE like STS, we receive this annotation as a gift. This annotation applies at the main class which has main () method. The Main class serves two purposes in a Spring Boot application: *configuration* and *bootstrapping*.

In fact *@SpringBootApplication* is a combination of three annotations with their default values. They are *@Configuration, @ComponentScan,* and *@EnableAutoConfiguration*. Therefore, we can also say that **@SpringBootApplication** is a 3-in-1 annotation. This is an important annotation in the context of our topic 'Spring Boot Annotations With Examples'.

@EnableAutoConfiguration: enables the auto-configuration feature of Spring Boot.
@ComponentScan: enables @Component scan on the package to discover and register compo

TOP

as beans in Spring's application Context.
@Configuration: allows to register extra beans in the context or imports additional configuration classes.

We can also use above three annotations separately in place of @SpringBootApplication if we want any customized behavior of them.

@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 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={WebSocketMessagingAutoConfiguration.class})
public class MyWebSocketApplication {
      public static void main(String[] args) {
         ...
      }
}
```

Use of 'excludeName' in *@EnableAutoConfiguration*

```
@Configuration
@EnableAutoConfiguration(excludeName = {"org.springframework.boot.autoconfigure.websocket.
```

TOP

```
public class MyWebSocketApplication {
      public static void main(String[] args) {
         ...
      }
}
```

@SpringBootConfiguration

This annotation is a part of Spring Boot Framework. However, *@SpringBootApplication* inherits from it. Therefore, If an application uses *@SpringBootApplication*, it is already using *@SpringBootConfiguration*. Moreover, It acts as an alternative to the *@Configuration* annotation. The primary difference is that *@SpringBootConfiguration* allows configuration to be automatically discovered. *@SpringBootConfiguration* indicates that the class provides configuration and also applied at the class level. Particularly, this is useful in case of unit or integration tests. For example, observe the below code:

```
@SpringBootConfiguration
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
    @Bean
    public IEmployeeService employeeService() {
       return new EmployeeServiceImpl();
    }
}
```

@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'.

TOP

```
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 {
```

TOP

```
    @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

TOP

@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")
```

Links to Other Annotations

As stated in the introduction section of our article 'Spring Boot Annotations With Examples', we will discuss all annotations that we generally use in a Spring Boot web Application. Below are the links to continue with 'Spring boot Annotations with Examples':

1) Spring Boot Bean Annotations With Examples

2) Spring Boot MVC & REST Annotations With Examples

3) Spring Boot Security, Scheduling and Transactions Annotations With Examples

4) Spring Boot Errors, Exceptions and AOP Annotations With Examples

Where can we use Annotations?

Apart from learning 'Spring Boot Annotations With Examples', its also important to know what are places to apply annotations.

We can apply annotations to the declarations of different elements of a program such as to the declarations of classes, fields, methods, and other program elements. By convention, we apply them just before the declaration of the element. Moreover, as of the Java 8 release, we can apply annotations to the use of types. For example, below code snippets demonstrate the usage of annotations:

↑ TOP

1) Class instance creation expression

```
new @Interned MyObject();
```

2) Type cast

```
myString = (@NonNull String) str;
```

3) implements clause

```
class UnmodifiableList<T> implements
    @Readonly List<@Readonly T> { ... }
```

4) Thrown exception declaration

```
void monitorTemperature() throws
    @Critical TemperatureException { ... }
```

This form of annotation is called a type annotation. For more information, see Type Annotations and Pluggable Type Systems.

However, these annotations are not in tradition at present, but in the future we can see them in the code. Since we are discussing about 'Spring Boot Annotations With Examples', we need to at least have the basic idea of them.

Spring Boot Bean Annotations With Examples
February 20, 2021
In "Spring Boot"

Spring Boot MVC REST Annotations With Examples
February 20, 2021
In "Spring Boot"

Spring Boot Errors and AOP Annotations With Examples
February 20, 2021

TOP