Spring Boot Bean Annotations With Examples

Spring Boot Java Spring by devs5003 February 28, 2021 3

In this article we will discuss on 'Spring Boot Bean Annotations with Examples'. Needless to say, these annotations play a crucial role in creating basic as well as enterprise level Spring Boot Applications. If you want to learn all annotations which are generally used in a Spring Boot Project, kindly visit our article 'Spring Boot Annotations with Examples'. Let's discuss about 'Spring Boot Bean Annotations with Examples' here only.

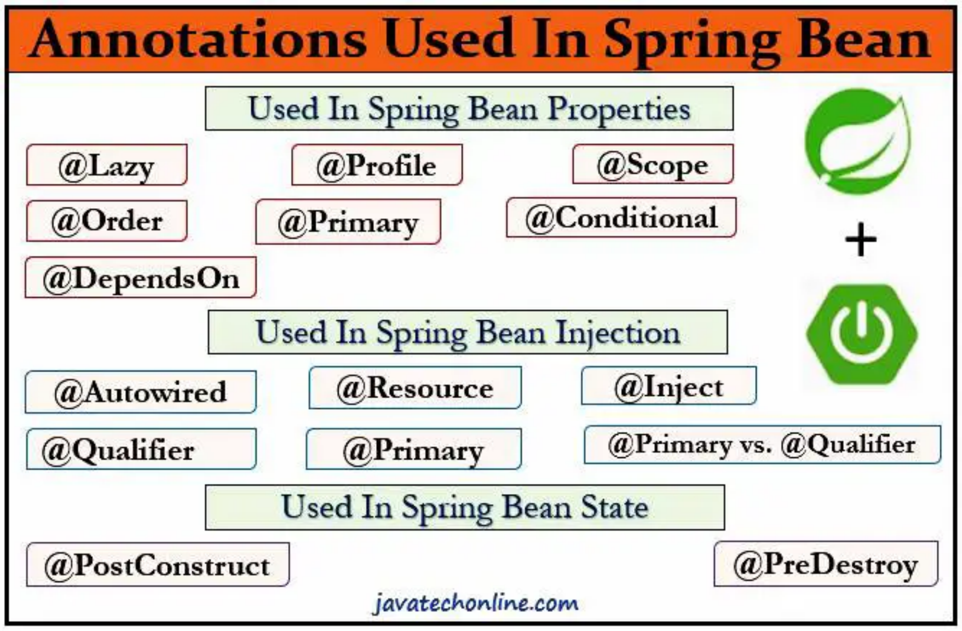

Table of Contents

What is a Java Bean?

Java Bean is a simple java helper class, used to transfer data between classes or applications. Typically, it doesn't act as a main class, but like postman or delivery boy between two classes. It doesn't contain any logic.

There are some standard guidelines to develop a Java Bean class:

1. We must declare Java Bean as a public class.
2. It is a recommendation to implement java.io.Serializable (used to sent data across network as we can easily send serializable data over the network). It is optional if you don't want to send data over network.
3. All member variables(bean properties) should be private & non-static (if we want to send data of 20 employees, 20 values will be sent. if its static only one value will be sent, so declaring non-static becomes mandatory to get full benefit of Java Bean)
4. Also Every bean property should have one setter and one getter method (Accessor Methods).

What is a Spring Bean or Component?

During Application startup, Spring instantiates objects and adds them to the Application Context. 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.

We have a separate detailed article on 'What is Spring Bean?', which can make your Spring Bean concept crystal clear.

What is @Bean annotation in Spring/Spring Boot?

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*. We use *@Bean* at method level. 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 actual bean. The annotated method produces a bean managed by the Spring IoC container.

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

For comparison sake, 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.

Annotations on Bean Properties

Here, we will be discussing annotations on Bean properties from our article 'Spring Boot Bean Annotations with Examples'. Let's start with @Lazy.

@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. Moreover, this is the equivalent for the XML based configuration's default-lazy-init="true" attribute. Beans that are not annotated with *@Lazy* are initialized eagerly. For example, below code demonstrates the use of *@Lazy* annotation:

```
@Configuration
public class AppConfig {
    @Lazy
    @Bean
    public Employee getEmployee() {
        return new Employee();
    }
}
```

As shown in the above example, in order to lazy load only specific beans, use *@Lazy* annotation along with *@Bean* annotation on a specific method.

@Profile

Before getting into *@Profile*, lets understand the concept of Profiles in Spring Boot. Spring Profiles provide a way to split parts of your application configuration and make it only available in certain environments. We all know that we have various environments in developing a project such as Development, Test, UAT, Production etc. In fact, we have specific configurations for a specific environment like hostname, port etc. Moreover, we need some specific bean to initialize only at a specific environment. Now let's observe how we can make a bean belong to a particular 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 initialized if the defined profiles are active. The annotation simply takes the names of one or multiple profiles. For example, let's consider we have a bean that should only be active during development but not deployed in production. We annotate that bean with a development profile, and it will only be present in the container during development. But in production, the development won't be active.

```
@Component
@Profile("development")
```

```
public class MyConfigClass{ }
```

In contrast, we can also prefix profile names with a NOT operator, e.g., *!*development, to exclude them from a profile. For example, we can initialize below component only if development profile is not active.

```
@Component
@Profile("!development")
public class MyConfigClass{ }
```

@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

Out of aforementioned six scopes, four are available only if we use a web-aware ApplicationContext. However, Singleton and prototype scopes are available in any type of IOC containers. To declare the scope of a bean we use *@Scope* annotation. If no scope is specified, Singleton scope is the default value. For example, to declare a prototype scope, we will use *@Scope* as below:

```
@Scope("prototype")
@Bean
public Student studentPrototype() {
    return new Student();
}
```

@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.

For Example, let's assume that we have two beans BeanA and BeanB and BeanB depends on BeanA. Therefore, our code will look like below:

```
@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. However the default value is Ordered.*LOWEST_PRECEDENCE*. This value indicates that the component has the lowest priority among all other ordered components. Similarly, the value Ordered.*HIGHEST_PRECEDENCE* indicates that the component has the highest priority among components. If we provide order value in @Order attribute, the lower number has the highest precedence ie. Order annotation with 1 will run before 2. We can even provide negative numbers. The order with the smaller value will always take precedence. If Component/Bean does not have *@Order*, by default it will execute in alphabetical order of bean name. Further, if two components are with the same order value, the alphabetical order of component name will win.

♥ **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

For example, let's observe below code:

```
@Component
@Order(-1)
public class A { }

@Component
@Order(5)
public class D { }
```

```
@Component
@Order(-24)
public class C { }

@Component
@Order(5)
public class B { }

@Component
public class F { }

@Component
public class E { }
```

From the above components, the order of execution will be : C, A, B, D, E, F

@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;
   }
}
```

Needless to say, in above EmployeeService class, PermanentEmployee will be injected via autowiring.

♥ ***Note :*** If we don't provide @Primary in PermanentEmployee component and try to run the application, Spring throws NoUniqueBeanDefinitionException. However, to access beans of the same type we generally use *@Qualifier("beanName")* annotation. In fact we apply it at the injection point along with *@Autowired.* In our case, as we selected the beans at the configuration phase so we can't apply *@Qualifier* annotation here.

@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. In short, annotated bean is created only if all conditions are satisfied. Spring 4.0 introduced this new annotation *@Conditional* that allows us to use either pre-defined or custom conditions that we can apply to bean in application context. For example, let's see one pre-defined condition with *@ConditionalOnJava*.

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

The above JavaBean will be loaded only if the running Java version in 9. Now let's see the most popular and commonly used pre-defined conditional annotation *@ConditionalOnProperty* in Spring Boot projects. It allows to load beans conditionally depending on a certain environment property:

```
@Configuration
@ConditionalOnProperty(
        value="module.enabled",
        havingValue = "true",
        matchIfMissing = true
        )
class MyPaymentModule {
...
}
```

The MyPaymentModule only loaded if the module.enabled property has the value true. If the property is not set at all, it will still be loaded, because we have defined matchIfMissing as true. This way, we have created a module that is loaded by default until we decide otherwise.

- *@ConditionalOnProperty*
- *@ConditionalOnExpression*
- *@ConditionalOnBean*
- *@ConditionalOnMissingBean*
- *@ConditionalOnResource*

The conditional annotations described above are the more common ones that we might use in any Spring Boot application. Spring Boot delivers even more conditional annotations. They are, however, not as common and some of them are more preferable for framework development rather than application development. Although Spring Boot framework uses some of them intensively.

- *@ConditionalOnClass*
- *@ConditionalOnMissingClass*
- *@ConditionalOnJndi*
- *@ConditionalOnJava*
- *@ConditionalOnSingleCandidate*
- *@ConditionalOnWebApplication*
- *@ConditionalOnNotWebApplication*
- *@ConditionalOnCloudPlatform*

Annotations on Bean Injection

Let's discuss next annotations on Spring Bean Injection from our article 'Spring Boot Bean Annotations with Examples'. Needless to say, how important they are in a Spring Boot Project.

@Autowired, @Resource, @Inject

These annotations belong to dependency injection. They provide classes with a declarative way to resolve dependencies. *@Resource* and *@Inject* belong to the Java extension packages javax.annotation.Resource and javax.inject.Inject respectively. However, the *@Autowired* annotation belongs to the org.springframework.beans.factory.annotation package. 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. We can use *@Autowired* on properties or fields, setters, and constructors. Let's see how

we can use *@Autowired* in a property. Let's define a bean as PaymentService.

```
@Service
public class PaymentService {

    public void doPayment() { .... }
}
```

Then, we'll inject this bean into the PaymentController bean using @Autowired on the field definition as below:

```
@Controller
public class PaymentController {
   @Autowired
   private PaymentService service;

   public void getPayment() {
        service.doPayment();
   }
}
```

Here, Spring Container injects PaymentService when PaymentController created.

@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

@Resource Execution Precedence

@Resource has execution paths as below listed by precedence:
First Match by Name
Then Match by Type
Then Match by Qualifier

@Inject Execution Precedence

This annotation has the same execution path preference as the *@Autowired* annotation:
First Match by Type

Then Match by Qualifier
Then Match by Name
Moreover, in order to access the *@Inject* annotation, the javax.inject library has to be declared as a Gradle or Maven dependency.

@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. For instance, let's see how we can use the *@Qualifier* annotation to indicate the required bean. Consider the below code:

```
public interface PaymentService { }

@Service
public class CardPaymentService implements PaymentService { }

@Service
public class CashPaymentService implements PaymentService { }

@Controller
public class PaymentController {

   @Autowired
   @Qualifier("cardPaymentService")
   private PaymentService service;

   public void getPayment() {
   service.doPayment();
   }
}
```

Here, By including the *@Qualifier* annotation together with the name of the specific implementation(cardPaymentService) we have eliminated the ambiguity as Spring container finds two beans of the same type.

@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 is 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. In this case, we should place the *@Primary* annotation in one of the implementing classes. For example, visit 'Annotations on Bean Properties' section of this article.

@Primary vs @Qualifier is also the important point from the interview point of view. As the interviewer will ask the difference between them when you will be explaining the concept of ambiguity in bean injection.

Annotations on Bean State

Now let's discuss annotations on Bean State from our article 'Spring Boot Bean Annotations with Examples'. The below two annotations are part of Spring Bean Life Cycle. 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");
        }
    }
```

FAQ

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.

Tagged @Autowired @autowired annotation @bean annotation @bean annotation in spring boot @bean annotation in spring boot example @bean annotation spring boot @bean example in spring boot @bean in spring boot example @component vs @bean @Conditional @DependsOn @Inject @Lazy @Order @PostConstruct @PreDestroy @Primary @primary and @qualifier in spring boot @primary in spring @Primary vs @Qualifier @Profile @Qualifier @qualifier annotation in spring boot example @qualifier in spring boot @qualifier vs @primary @required in spring @Resource @Scope @service vs @component autowired annotation in spring boot autowired can be used at which level autowired java bean annotation in spring bean in spring bean in spring boot bean spring boot difference between @bean and @component Difference between @Primary and @Qualifier difference between @primary and @qualifier in spring difference between @Qualifier and @Primary inject vs autowired java bean annotation java spring boot bean example primary annotation qualifier annotation qualifier annotation in spring qualifier annotation in spring boot qualifier example qualifier spring boot required annotation in spring spring bean spring bean annotation spring bean example spring bean java spring boot @bean annotation spring boot @bean example spring boot @qualifier spring boot annotation spring boot annotations interview questions spring boot bean Spring Boot Bean Annotations With Examples spring boot bean injection spring boot create bean spring core annotations springboot bean annotation what is @bean annotation in spring boot what is a bean in java what is bean in java spring what is bean in spring what is bean in spring boot what is the use of @bean annotation in spring

‹ Previous article
Spring Boot MVC REST Annotations With Examples

Next article ›
Spring Boot Annotations With Examples