**2. DI-Related Annotations**

**2.1. *@Autowired***

We can use the *@Autowired* to **mark a dependency which Spring is going to resolve and inject**. We can use this annotation with a constructor, setter, or field injection.

Constructor injection:

**class** Car {

Engine engine;

@Autowired

Car(Engine engine) {

**this**.engine = engine;

}

}

Setter injection:

**class** Car {

Engine engine;

@Autowired

**void** setEngine(Engine engine) {

**this**.engine = engine;

}

}

Field injection:

**class** Car {

@Autowired

Engine engine;

}

*@Autowired* has a *boolean* argument called *required* with a default value of *true*. It tunes Spring's behavior when it doesn't find a suitable bean to wire. When *true*, an exception is thrown, otherwise, nothing is wired.

Note, that if we use constructor injection, all constructor arguments are mandatory.

Starting with version 4.3, we don't need to annotate constructors with *@Autowired* explicitly unless we declare at least two constructors.

For more details visit our articles about [*@Autowired*](https://www.baeldung.com/spring-autowire) and [constructor injection](https://www.baeldung.com/constructor-injection-in-spring).

**2.2. *@Bean***

*@Bean* marks a factory method which instantiates a Spring bean:

@Bean

Engine engine() {

**return** **new** Engine();

}

**Spring calls these methods** when a new instance of the return type is required.

The resulting bean has the same name as the factory method. If we want to name it differently, we can do so with the *name* or the *value* arguments of this annotation (the argument *value* is an alias for the argument *name*):

@Bean("engine")

Engine getEngine() {

**return** **new** Engine();

}

Note, that all methods annotated with *@Bean* must be in *@Configuration* classes.

**2.3. *@Qualifier***

We use*@Qualifier* along with *@Autowired* to **provide the bean id or bean name** we want to use in ambiguous situations.

For example, the following two beans implement the same interface:

**class** Bike **implements** Vehicle {}

**class** Car **implements** Vehicle {}

If Spring needs to inject a *Vehicle* bean, it ends up with multiple matching definitions. In such cases, we can provide a bean's name explicitly using the *@Qualifier* annotation.

Using constructor injection:

@Autowired

Biker(@Qualifier("bike") Vehicle vehicle) {

**this**.vehicle = vehicle;

}

Using setter injection:

@Autowired

**void** setVehicle(@Qualifier("bike") Vehicle vehicle) {

**this**.vehicle = vehicle;

}

Alternatively:

@Autowired

@Qualifier("bike")

**void** setVehicle(Vehicle vehicle) {

**this**.vehicle = vehicle;

}

Using field injection:

@Autowired

@Qualifier("bike")

Vehicle vehicle;

For a more detailed description, please read [this article](https://www.baeldung.com/spring-autowire).

**2.4. *@Required***

*@Required* on setter methods to mark dependencies that we want to populate through XML:

@Required

**void** setColor(String color) {

**this**.color = color;

}

<**bean** class=**"com.baeldung.annotations.Bike"**>

<**property** name=**"color"** value=**"green"** />

</**bean**>

Otherwise, *BeanInitializationException* will be thrown.

**2.5. *@Value***

We can use [*@Value*](https://www.baeldung.com/spring-value-annotation) for injecting property values into beans. It's compatible with constructor, setter, and field injection.

Constructor injection:

Engine(@Value("8") **int** cylinderCount) {

**this**.cylinderCount = cylinderCount;

}

Setter injection:

@Autowired

**void** setCylinderCount(@Value("8") **int** cylinderCount) {

**this**.cylinderCount = cylinderCount;

}

Alternatively:

@Value("8")

**void** setCylinderCount(**int** cylinderCount) {

**this**.cylinderCount = cylinderCount;

}

Field injection:

@Value("8")

**int** cylinderCount;

Of course, injecting static values isn't useful. Therefore, we can use **placeholder strings** in *@Value* to wire values **defined in external sources**, for example, in *.properties* or *.yaml* files.

Let's assume the following *.properties* file:

engine.fuelType=petrol

We can inject the value of *engine.fuelType* with the following:

@Value("${engine.fuelType}")

String fuelType;

We can use *@Value* even with SpEL. More advanced examples can be found in our [article about *@Value*](https://www.baeldung.com/spring-value-annotation).

**2.6. *@DependsOn***

We can use this annotation to make Spring **initialize other beans before the annotated one**. Usually, this behavior is automatic, based on the explicit dependencies between beans.

We only need this annotation **when the dependencies are implicit**, for example, JDBC driver loading or static variable initialization.

We can use *@DependsOn* on the dependent class specifying the names of the dependency beans. The annotation's *value* argument needs an array containing the dependency bean names:

@DependsOn("engine")

**class** Car **implements** Vehicle {}

Alternatively, if we define a bean with the *@Bean* annotation, the factory method should be annotated with *@DependsOn*:

@Bean

@DependsOn("fuel")

Engine engine() {

**return** **new** Engine();

}

**2.7. *@Lazy***

We use [*@Lazy*](https://www.baeldung.com/spring-lazy-annotation) when we want to initialize our bean lazily. By default, Spring creates all singleton beans eagerly at the startup/bootstrapping of the application context.

However, there are cases when**we need to create a bean when we request it, not at application startup**.

This annotation behaves differently depending on where we exactly place it. We can put it on:

* a *@Bean* annotated bean factory method, to delay the method call (hence the bean creation)
* a @*Configuration* class and all contained *@Bean* methods will be affected
* a *@Component* class, which is not a *@Configuration* class, this bean will be initialized lazily
* an *@Autowired* constructor, setter, or field, to load the dependency itself lazily (via proxy)

This annotation has an argument named *value* with the default value of *true*. It is useful to override the default behavior.

For example, marking beans to be eagerly loaded when the global setting is lazy, or configure specific *@Bean* methods to eager loading in a *@Configuration* class marked with *@Lazy*:

@Configuration

@Lazy

**class** VehicleFactoryConfig {

@Bean

@Lazy(false)

Engine engine() {

**return** **new** Engine();

}

}

For further reading, please visit [this article](https://www.baeldung.com/spring-lazy-annotation).

**2.8. *@Lookup***

A method annotated with *@Lookup* tells Spring to return an instance of the method’s return type when we invoke it.

Detailed information about the annotation [can be found in this article](https://www.baeldung.com/spring-lookup).

**2.9. *@Primary***

Sometimes we need to define multiple beans of the same type. In these cases, the injection will be unsuccessful because Spring has no clue which bean we need.

We already saw an option to deal with this scenario: marking all the wiring points with *@Qualifier* and specify the name of the required bean.

However, most of the time we need a specific bean and rarely the others. We can use *@Primary* to simplify this case: if **we mark the most frequently used bean with *@Primary*** it will be chosen on unqualified injection points:

@Component

@Primary

**class** Car **implements** Vehicle {}

@Component

**class** Bike **implements** Vehicle {}

@Component

**class** Driver {

@Autowired

Vehicle vehicle;

}

@Component

**class** Biker {

@Autowired

@Qualifier("bike")

Vehicle vehicle;

}

In the previous example *Car* is the primary vehicle. Therefore, in the *Driver* class, Spring injects a *Car* bean. Of course, in the *Biker* bean, the value of the field *vehicle* will be a *Bike* object because it's qualified.

**2.10. *@Scope***

We use *@Scope* to define the [scope](https://www.baeldung.com/spring-bean-scopes) of a *@Component* class or a *@Bean* definition*.*It can be either *singleton, prototype, request, session, globalSession* or some custom scope.

For example:

@Component

@Scope("prototype")

**class** Engine {}

**3. Context Configuration Annotations**

We can configure the application context with the annotations described in this section.

**3.1. *@Profile***

If we want Spring to **use a *@Component* class or a *@Bean* method only when a specific profile is active**, we can mark it with *@Profile*. We can configure the name of the profile with the *value* argument of the annotation:

@Component

@Profile("sportDay")

**class** Bike **implements** Vehicle {}

You can read more about profiles in [this article](https://www.baeldung.com/spring-profiles).

**3.2. *@Import***

We can use **specific *@Configuration* classes without component scanning** with this annotation. We can provide those classes with *@Import*‘s *value* argument:

@Import(VehiclePartSupplier.class)

**class** VehicleFactoryConfig {}

**3.3. *@ImportResource***

We can **import XML configurations** with this annotation. We can specify the XML file locations with the *locations* argument, or with its alias, the *value* argument:

@Configuration

@ImportResource("classpath:/annotations.xml")

**class** VehicleFactoryConfig {}

**3.4. *@PropertySource***

With this annotation, we can **define property files for application settings**:

@Configuration

@PropertySource("classpath:/annotations.properties")

**class** VehicleFactoryConfig {}

*@PropertySource* leverages the Java 8 repeating annotations feature, which means we can mark a class with it multiple times:

@Configuration

@PropertySource("classpath:/annotations.properties")

@PropertySource("classpath:/vehicle-factory.properties")

**class** VehicleFactoryConfig {}

**3.5. *@PropertySources***

We can use this annotation to specify multiple *@PropertySource*configurations:

@Configuration

@PropertySources({

@PropertySource("classpath:/annotations.properties"),

@PropertySource("classpath:/vehicle-factory.properties")

})

**class** VehicleFactoryConfig {}

Note, that since Java 8 we can achieve the same with the repeating annotations feature as described above.

**4. Conclusion**

In this article, we saw an overview of the most common Spring core annotations. We saw how to configure bean wiring and application context, and how to mark classes for component scanning.