(/)

Guide to Spring @Autowired

Last modified: January 7, 2021

by baeldung (https://www.baeldung.com/author/baeldung/)

Spring (https://www.baeldung.com/category/spring/) +

Spring Core Basics (https://www.baeldung.com/tag/spring-core-basics/)

Spring DI (https://www.baeldung.com/tag/spring-di/)

1. Overview

Starting with Spring 2.5, the framework introduced annotations-driven *Dependency Injection*. The main annotation of this feature is *@Autowired.* **It allows Spring to resolve and inject collaborating beans into our bean.**

Further reading:

Spring Component Scanning (/spring-component-scanning)

Learn about the mechanism behind Spring component scanning, and how you can tweak it to your own needs

Read more (/spring-component-scanning) →

Intro to Inversion of Control and Dependency Injection with Spring (/inversion-control-and-dependency-injection-in-spring)

A quick introduction to the concepts of Inversion of Control and Dependency Injection, followed by a simple demonstration using the Spring Framework

Read more (/inversion-control-and-dependency-injection-in-spring) →

In this tutorial, we'll first take a look at how to enable autowiring and the various ways to autowire beans. Afterward, we'll talk about **resolving bean conflicts using *@Qualifier* annotation,** as well as potential exception scenarios.

2. Enabling *@Autowired* Annotations

The Spring framework enables automatic dependency injection. In other words, **by declaring all the bean dependencies in a Spring configuration file, Spring container can autowire relationships between collaborating beans**. This is called ***Spring bean autowiring***.

To use Java-based configuration in our application, let's enable annotation-driven injection to load our Spring configuration:

```
@Configuration
@ComponentScan("com.baeldung.autowire.sample")
public class AppConfig {}
```

Alternatively, the *<context:annotation-config>* annotation (/spring-contextannotation-contextcomponentscan#:~:text=The%20%3Ccontext%3Aannotation%2Dconfig,annotation%2Dconfig%3E%20can%20resolve.) is mainly used to activate the dependency injection annotations in Spring XML files.

Moreover, **Spring Boot introduces the *@SpringBootApplication* (/spring-boot-annotations#spring-boot-application)annotation**. This single annotation is equivalent to using *@Configuration*, *@EnableAutoConfiguration*, and *@ComponentScan*.

Let's use this annotation in the main class of the application:

```
@SpringBootApplication
class VehicleFactoryApplication {
    public static void main(String[] args) {
        SpringApplication.run(VehicleFactoryApplication.class, args);
    }
}
```

As a result, when we run this Spring Boot application, **it will automatically scan the components in the current package and its sub-packages**. Thus it will register them in Spring's Application Context, and allow us to inject beans using *@Autowired*.

3. Using *@Autowired*

After enabling annotation injection, **we can use autowiring on properties, setters, and constructors**.

3.1. *@Autowired* on Properties

Let's see how we can annotate a property using *@Autowired*. This eliminates the need for getters and setters.

First, let's define a *fooFormatter* bean:

```
@Component("fooFormatter")
public class FooFormatter {
    public String format() {
        return "foo";
    }
}
```

Then, we'll inject this bean into the *FooService* bean using *@Autowired* on the field definition:

```
@Component
public class FooService {
    @Autowired
    private FooFormatter fooFormatter;
}
```

As a result, Spring injects *fooFormatter* when *FooService* is created.

3.2. *@Autowired* on Setters

Now let's try adding *@Autowired* annotation on a setter method.

In the following example, the setter method is called with the instance of *FooFormatter* when *FooService* is created:

```
public class FooService {
    private FooFormatter fooFormatter;
    @Autowired
    public void setFooFormatter(FooFormatter fooFormatter) {
        this.fooFormatter = fooFormatter;
    }
}
```

3.3. *@Autowired* on Constructors

Finally, let's use *@Autowired* on a constructor.

We'll see that an instance of *FooFormatter* is injected by Spring as an argument to the *FooService* constructor:

```
public class FooService {
    private FooFormatter fooFormatter;
    @Autowired
    public FooService(FooFormatter fooFormatter) {
        this.fooFormatter = fooFormatter;
    }
}
```

4. *@Autowired* and Optional Dependencies

When a bean is being constructed, the *@Autowired* dependencies should be available. Otherwise, **if Spring cannot resolve a bean for wiring, it will throw an exception**.

Consequently, it prevents the Spring container from launching successfully with an exception of the form:

```
Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:
No qualifying bean of type [com.autowire.sample.FooDAO] found for dependency:
expected at least 1 bean which qualifies as autowire candidate for this dependency.
Dependency annotations:
{@org.springframework.beans.factory.annotation.Autowired(required=true)}
```

To fix this, we need to declare a bean of the required type:

```
public class FooService {
    @Autowired(required = false)
    private FooDAO dataAccessor;
}
```

5. Autowire Disambiguation

By default, Spring resolves *@Autowired* entries by type. **If more than one bean of the same type is available in the container, the framework will throw a fatal exception**.

To resolve this conflict, we need to tell Spring explicitly which bean we want to inject.

5.1. Autowiring by *@Qualifier*

For instance, let's see how we can use the *@Qualifier* (/spring-qualifier-annotation) annotation to indicate the required bean.

First, we'll define 2 beans of type *Formatter*:

```
@Component("fooFormatter")
public class FooFormatter implements Formatter {
    public String format() {
        return "foo";
    }
}
```

```
@Component("barFormatter")
public class BarFormatter implements Formatter {
    public String format() {
        return "bar";
    }
}
```

Now let's try to inject a *Formatter* bean into the *FooService* class:

```
public class FooService {
    @Autowired
    private Formatter formatter;
}
```

In our example, there are two concrete implementations of *Formatter* available for the Spring container. As a result, **Spring will throw a *NoUniqueBeanDefinitionException* exception when constructing the *FooService***:

```
Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [com.autowire.sample.Formatter] is defined:
expected single matching bean but found 2: barFormatter,fooFormatter
```

We can avoid this by narrowing the implementation using a *@Qualifier* annotation:

```
public class FooService {
    @Autowired
    @Qualifier("fooFormatter")
    private Formatter formatter;
}
```

When there are multiple beans of the same type, it's a good idea to **use *@Qualifier* to avoid ambiguity.**

Please note that the value of the *@Qualifier* annotation matches with the name declared in the *@Component* annotation of our *FooFormatter* implementation.

5.2. Autowiring by Custom Qualifier

Spring also allows us to **create our own custom *@Qualifier* annotation**. To do so, we should provide the *@Qualifier* annotation with the definition:

```
@Qualifier
@Target({
  ElementType.FIELD, ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface FormatterType {
    String value();
}
```

Then we can use the *FormatterType* within various implementations to specify a custom value:

```
@FormatterType("Foo")
@Component
public class FooFormatter implements Formatter {
    public String format() {
        return "foo";
    }
}
```

```
@FormatterType("Bar")
@Component
public class BarFormatter implements Formatter {
    public String format() {
        return "bar";
    }
}
```

Finally, our custom Qualifier annotation is ready to use for autowiring:

```
@Component
public class FooService {
    @Autowired
    @FormatterType("Foo")
    private Formatter formatter;
}
```

The value specified in the ***@Target* meta-annotation restricts where to apply the qualifier,** which in our example is fields, methods, types, and parameters.

5.3. Autowiring by Name

Spring uses the bean's name as a default qualifier value. It will inspect the container and look for a bean with the exact name as the property to autowire it.

Hence, in our example, Spring matches the *fooFormatter* property name to the *FooFormatter* implementation. Therefore, it injects that specific implementation when constructing *FooService*:

```
public class FooService {
 @Autowired
private Formatter fooFormatter;
}
```

6. Conclusion

In this article, we discussed autowiring and the different ways to use it. We also examined ways to solve two common autowiring exceptions caused by either a missing bean or an ambiguous bean injection.

The source code of this article is available on the GitHub project (https://github.com/eugenp/tutorials/tree/master/spring-core-2).

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)

JACKSON JSON TUTORIAL (/JACKSON)

HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)

REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)

SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)

SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE COURSES (HTTPS://COURSES.BAELDUNG.COM)

JOBS (/TAG/ACTIVE-JOB/)

THE FULL ARCHIVE (/FULL_ARCHIVE)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS)

ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)