**1. Overview**

Starting with Spring 2.5, the framework introduced a new style of Dependency Injection driven by @Autowired Annotations. This annotation allows spring to resolve and inject collaborating beans into your bean.

In this tutorial, we will look at how to enable autowiring, various ways to wire in beans, making beans optional, and resolving bean conflicts using @Qualifier annotation along with potential exception scenarios.

**2. Enabling @Autowired Annotations**

If you are using Java based configuration in your application you can enable annotation-driven injection by using AnnotationConfigApplicationContext to load your spring configuration as below:

@Configuration

@ComponentScan ("com.baeldung.autowire.sample")

public class AppConfig {}

As an alternative, in Spring XML, it can be enabled by declaring it in Spring XML files like so: <context:annotation-config/>

**3. Using @Autowired**

Once annotation injection is enabled, autowiring can be used on properties, setters, and constructors.

**3.1. @Autowired on Properties**

The annotation can be used directly on properties, therefore eliminating the need for getters and setters:

@Component ("fooFormatter")

public class FooFormatter {

public String format() {

return "foo";

}

}

@Component

public class FooService {

@Autowired

private FooFormatter fooFormatter;

}

In the above example, spring looks for and injects fooFormatter when FooService is created.

3.2. @Autowired on Setters

The @Autowired annotation can be used on setter methods. In the below example, when the annotation is used on the setter method, 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**

The @Autowired annotation can also be used on constructors. In the below example, when the annotation is used on a constructor, an instance of FooFormatter is injected as an argument to the constructor when FooService is created:

public class FooService {

private FooFormatter fooFormatter;

@Autowired

public FooService(FooFormatter fooFormatter) {

this.fooFormatter = fooFormatter;

}

}

**4. @Autowired and Optional Dependencies**

Spring expects @Autowired dependencies to be available when the dependent bean is being constructed. If the framework cannot resolve a bean for wiring, it will throw the below-quoted exception and prevent the spring container from launching successfully:

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 avoid this from happening, a bean can optional be specified as below:

public class FooService {

@Autowired (required = false)

private FooDAO dataAccessor;

}

5. Autowire Disambiguation

By default, spring resolves @Autowired entries by type. If more than one beans of the same type are available in the container, the framework will throw a fatal exception indicating that more than one bean is available for autowiring.

**5.1. Autowiring by @Qualifier**

The @Qualifier annotation can be used to hint at and narrow down the required bean:

@Component ("fooFormatter")

public class FooFormatter implements Formatter {

public String format() {

return "foo";

}

}

@Component ("barFormatter")

public class BarFormatter implements Formatter {

public String format () {

return "bar";

}

}

public class FooService {

@Autowired

private Formatter formatter;

}

Since there are two concrete implementations of Formatter available for the spring container to inject, 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

This can be avoided by narrowing the implementation using a @Qualifier annotation:

public class FooService {

@Autowired

@Qualifier("fooFormatter")

private Formatter formatter;

}

By specifying the @Qualifier with the name of the specific implementation, in this case as fooFormatter, we can avoid ambiguity when spring finds multiple beans of the same type.

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 allows us to create our own @Qualifier annotation. To create a custom Qualifier, define an annotation and provide the @Qualifier annotation within the definition as below:

@Qualifier

@Target ({

ElementType.FIELD, ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER})

@Retention (RetentionPolicy.RUNTIME)

public @interface FormatterType {

String value ();

}

Once defined, the FormatterType can be used within various implementations to specify 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";

}

}

Once the implementations are annotated, the custom Qualifier annotation can be used as below:

@Component

public class FooService {

@Autowired

@FormatterType ("Foo")

private Formatter formatter;

}

The value specified in the @Target annotation restrict where the qualifier can be used to mark injection points.

In the above code snippet, the qualifier can be used to disambiguate the point where spring can inject the bean into a field, a method, a type, and a parameter.

**5.2. Autowiring by Name**

As a fall-back spring uses the bean name as a default qualifier value.

So by defining the bean property name, in this case as fooFormatter, spring matches that to the FooFormatter implementation and injects that specific implementation when FooService is constructed:

public class FooService {

@Autowired

private Formatter fooFormatter;

}

**6. Conclusion**

Although both @Qualifier and bean name fallback match can be used to narrow down to a specific bean, autowiring is really all about injection by type and this is how best to use this container feature.

The source code of this tutorial can be found in the GitHub project – this is an Eclipse based project, so it should be easy to import and run as it is.