Spring Labs

Labs Spring

# Wiring Dependencies using XML

In this lab you will learn to use the Spring container to inject dependencies. Dependency Injection can be configured using XML or annotations. In this lab you will use XML.

## Step 1 - Examine the starter project

Examine the starter project.

All Spring dependencies are included in the POM file of the parent Maven project. This parent project is the parent for the other 3 projects. This is the preferred way to work with Spring dependencies. Explore the POM file of the lab01 project to understand how the project is configured.

Start with the lab01 project. First add a Spring Bean Configuration File using the context menu in STS to the src/main/resources directory. Name the configuration file “applicationContext.xml”. Add a new class with a test method you will bootstrap the Spring container from code. To bootstrap Spring you’ll have to create an ApplicationContext. Use the following code to do so.

ApplicationContext applicationContext = new

ClassPathXmlApplicationContext("applicationContext.xml");

## Step 2 - Adding a service

Create a new Java interface “PrinterService” with a single method:

void print(String message);

Create a new class “ConsolePrinter” and implement the PrinterService interface. Now add a bean definition in the applicationContext.xml:

<bean id="printer" class="lab1.ConsolePrinter"/>

Test your configuration by looking up the printer bean in the applicationContext in the test class:

ApplicationContext applicationContext =

new ClassPathXmlApplicationContext("applicationContext.xml");

PrinterService printer =

applicationContext.getBean(PrinterService.class);

printer.print("Hallo!");

**Step 3 - Wiring dependencies**

Now you’ll add more beans that will depend on each other to see the different forms of dependency injection. Start by adding a new class “SetterDI” to the project. Add a property (private field and setter method) of type PrinterService to the class. This dependency must be wired by the Spring container, you’ll do that in a minute. Also add a method “sayHello” to the class that uses the PrinterService to print a message to the console.

Now add the new class to the Spring configuration as you did in the previous step. Test the project by looking up the setterDI bean from the test method of the project and calling the sayHello method. This should result in a *NullPointerException* because the PrinterService is not wired to the Hello bean yet.

To wire the PrinterService dependency you will use the most basic form of dependency injection by configuring the dependency in XML. Change your bean definition as follows:

<bean id="setterDI" class="lab1.SetterDI">

<property name="printer" ref="printer"/>

</bean>

This is *setter injection.* Spring will use the setPrinter method to inject the printer bean into the hello bean.

## Step 4 - Constructor injection

Instead of using *setter injection* you could use *constructor injection*. Add a new class to the project named “ConstructorDI”. Add a private field “printer” of type PrinterService but omit get/set methods. Add a constructor with an argument of type PrinterService to the class and initialize the printer field using this argument. Again, add a method “sayHello” that uses the printer.

Add a bean declaration to the configuration for the new class. To wire the PrinterService dependency you’ll now have to use a *constructor-arg*.

<bean id="constructorDI" class="lab1.ConstructorDI">

<constructor-arg ref="printer"/>

</bean>

Test the new class again by looking it up in the container from the test method.

## Step 5 - AutoWiring

In this step you’ll try out AutoWiring of dependencies. AutoWiring can be enabled on applicationlevel and on beanlevel, in this case you will enable AutoWiring for a single bean. Add a new class “AutoWireDI” to the project and add a private field of type PrinterService, a setter method for this field and a sayHello method.

Add the bean configuration to the applicationContext. Instead of wiring the dependency to the PrinterService explicitly you will autowire this dependency using the following configuration:

<bean id="autowiringDI" class="lab1.AutoWireDI" autowire="byType"/>

Add code to the test method to test the new bean. Also try out the different modes of autowiring (byType, byName etc.).

## Step 6 - Factory methods

In this step you will use a factory method to initialize a bean. Create a new class “FactoryMethodInstantiation” to the project. Add a private no-arg constructor so you can be sure Spring will not call this constructor. Also add two private fields, a PrinterService and a String.

Now add a factory method. A factory method must be static and return an instance of the class it’s part of. The factory method should have two arguments, a PrinterService instance and a String. Make sure the method looks as follows:

public static FactoryMethodInstantiation createInstance(

PrinterService printer, String value) {

FactoryMethodInstantiation instance =

new FactoryMethodInstantiation();

instance.printer = printer;

instance.value = value;

return instance;

}

Add a printValue method to the class that uses the printer instance to print the value field to the console.

Configure the new bean in the application context. To pass arguments to a factory method you’ll have to use constructor-args (although it’s not really a constructor).

<bean id="factoryMethodInstantiation" class="lab1.FactoryMethodInstantiation"

factory-method="createInstance">

<constructor-arg index="0" ref="printer" />

<constructor-arg index="1" value="MyTest" />

</bean>

Test the bean again from the project’s test method.

## Step 7 - Factory beans

The factory approach can be taken a step further by implementing the factory-method on a separate factory-bean instead of in the class itself. This is similar to the distinction of the Factory Method and Abstract Factory pattern. You can also use factory beans to instantiate and fill lists of data for example. That’s what you’ll do in this step.

Add a new class “NameFactory” to the project. Add a non-static method createNameList to the class that returns a java.util.List of names.

For this approach you’ll need two bean configurations in the applicationContext. One of the bean that represents the list of names, and one for the factory.

<bean id="nameFactory" class="lab1.NameFactory"/>

<bean id="names" factory-bean="nameFactory"

factory-method="createNameList"/>

Test the configuration by looking up the names bean from the test method and print it to the console.

## Step 8 - Scopes

So far you have only used the default singleton scope. To understand the difference between singleton and prototype scope, and to understand method injection you’ll experiment with those in this step. First of all create a new class “HitCounter” to the project and add a field “hits” of type int to it. Also add a void method “increment” that increments the hits field. Configure the class as a bean in the applicationContext.

Add another class “Hitter” to the project. This class should have a dependency to the HitCounter class. Add a void method “hit” to the class that calls the HitCounter’s increment method. Now add *two* bean definitions for the Hitter class (meaning two instances of the same class). The configuration should now be as follows.

<bean id="hitCounter" class="lab1.HitCounter"/>

<bean id="hitter1" class="lab1.Hitter">

<property name="counter" ref="hitCounter"/>

</bean>

<bean id="hitter2" class="lab1.Hitter">

<property name="counter" ref="hitCounter"/>

</bean>

Retrieve both hitter1 and hitter2 from the applicationContext in the project’s test method. Call the hit method twice on both hitters. Call and print the getHits on hitter1. You should see 4 hits, because at this point the HitCounter is shared (it’s a singleton) by the two hitters.

Change the scope of the hitCounter bean to prototype. Test again. How many hits do you see? Each bean that references the hitCounter gets it’s own instance now.

What if you would want a new hitCounter instance on each call to the hitter’s hit method? This would not be very useful in this example, but it’s interesting anyway. You could use method injection for this. Add a new class “DynamicHitter” to the project and give it an abstract method “getHitCounter”, a “hit” and “getHits” method.

public abstract class DynamicHitter {

public void hit() {

getHitCounter().increment();

}

public int getHits() {

return getHitCounter().getHits();

}

public abstract HitCounter getHitCounter();

}

The abstract method will be implemented at runtime by Spring. To enable this you’ll need some extra configuration in the bean definition.

<bean id="hitter3" class="lab1.DynamicHitter">

<lookup-method name="getHitCounter" bean="hitCounter"/>

</bean>

Test the new bean by calling the hit method multiple times and asking for getHits after that. The result should be 0 hits, because a HitCounter is created by Spring each time the getHitCounter method is called.

# Wiring Dependencies using Java Based Configuration

## Step 1 – Lab02

Add a new class with a test method you will bootstrap the Spring container from code. To bootstrap Spring you’ll have to create an ApplicationContext. Use the following code to do so.

new AnnotationConfigApplicationContext(JavaBasedConfiguration.class);

Repeat the steps 1 to 8 of the first lab.

What is your opinion of Java Based Configuration in comparison to xml configuration.

# Wiring Dependencies using annotations

In this lab you will learn to use Dependency Injection using the different annotations available in the Spring framework. You will implement a simple MovieLister class that uses multiple MovieCatalogs to list titles of movies.

## Step 1 – Lab03

Add a new class with a test method you will bootstrap the Spring container from code. To bootstrap Spring you’ll have to create an ApplicationContext. Use the following code to do so. Add a new context file to src/main/resources. There are two steps in enabling an annotation based approach: Enable annotation configuration and enable component scanning. Component scanning scans the classpath for @Component annotated classes that will be added to the context. Annotation configuration is in the “context” namespace. The full configuration file is as follows:

<beans xmlns="<http://www.springframework.org/schema/beans>"

xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"

xmlns:context="<http://www.springframework.org/schema/context>"

xsi:schemaLocation="<http://www.springframework.org/schema/beans>

<http://www.springframework.org/schema/beans/spring-beans-3.0.xsd>

<http://www.springframework.org/schema/context>

[http://www.springframework.org/schema/context/spring-context-4.3.xsd](http://www.springframework.org/schema/context/spring-context-3.0.xsd)">

<context:annotation-config />

<context:component-scan base-package="lab2"/>

</beans>

Create a Tester class with a test method that bootstraps the Spring container.

## Step 2 - A simple movie lister

You’ll start with some basic annotation based dependency injection. First create a new class MovieLister. Add a void method listMovies to the class. Annotate the class with @Component to make it a Spring bean.

Add an interface MovieCatalog with a single method getMovieTitles that returns a List<String>. Implement the interface in a class GeneralMovieCatalog and return a few titles of movies and annotate this class too with @Component. Go back to the MovieLister class. Add a private field of type MovieCatalog, and use this field to print the list of movie titles returned by the getMovieCatalog method. To wire the dependency to the MovieCatalog you can use the @AutoWire annotation.

@Autowired

private MovieCatalog movieCatalog;

This is dependency injection by type, so there may only be one implementation of the MovieCatalog interface. Test if the dependency is wired correctly.

## Step 3 - Qualifiers

Create another implementation of the MovieCatalog interface named ActionMovieCatalog. When you try to start the application again the container should give an error telling that there is no unique bean of type MovieCatalog. The container can’t know which of the two implementations of the MovieCatalog interface should be injected. By using qualifiers you can bind a specific implementation class to an injection point, while still programming to interfaces at the Java level (it’s still easy to unit test).

You’ll create a fully type-safe qualifier instead of relying on Strings. While this is slightly more work, it can prevent errors. Create a new annotion named Action and annotate it as a Qualifier.

@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Action { }

Add the newly created @Action both on the ActionMovieCatalog class definition and the field where you want to inject the movie catalog.

@Autowired

private MovieCatalog movieCatalog;

Test the application again. Only action movies should be listed now.

## Step 4 - Multiple catalogs

In some cases you want to get all implementation of a certain interface. You can do that by injecting into a List or Map of that interface.

Add another field to the MovieLister that contains all movie catalogs including their bean names:

@Autowired

private Map<String, MovieCatalog> allCatalogs;

public void listMovies() {

for(String key : allCatalogs.keySet()) {

System.out.println(key);

for(String title : allCatalogs.get(key).getMovieCatalog()) {

System.out.println("-" + title);

}

}

}