Spring Container workshop

Objectives

In this workshop, we are going to create a simple application that prints out the message of the day. The goal is to work with the containers. We will create an XML file and configure the class that the client needs to instantiate and call its methods.

Recap

Earlier we talked about coupling and we saw that to reduce the coupling we need to do three things and this is not enough to do one or two of them. We should do all the three steps together to reduce the coupling in our code. These were the three steps:

1. Program to Interfaces
2. Dependency Injection
3. Centralise Configuration

We have done steps 1 and 2 in the previous chapter. Now we want to see how we can use a centralized configuration by using containers.

We will have a very simple example to see how the spring container works:

A new project

Download the *MessageProject* code from here.
In this project we will have a client that could be a web page, a java class with a main method. The main method will call a class *MessageOfTheDayService* which returns the message of the day.

Add the project to your IDE.
In the lib folder, there are 5 jar files, select all the jar files and add to the build path.

We write the interface first:

```
Interface:
```

```
public interface MessageOfTheDayService {
	public String getTodaysMessage();

}
```

And then we write an implementation for this interface:

```
public class MessageOfTheDayBasicImpl implements MessageOfTheDayService {

	private String message;

	public void setMessage(String message) {
		this.message= message;
	}
	@Override
	public String getTodaysMessage() {
		return message;
	}
}
```

The general pattern for working in spring

- The Client instead of creating the MessagOfTheDayService itself will get the object from the Spring container.
- In spring we call the objects *beans*.
- The spring creates the service object from a config file which is written in an xml file.
- Spring container will look at this xml file and based on what it finds in the file, will create an instance of MessageOfTheDayService.
- In the config file we will set the implementation class of the interface.
- This config file has the properties of the objects. For example message property in this example. This message will vary depending on the day. So it is a dependency.
- Spring will call the setMessage method and pass in the data from the external configuration file into the object. So the client will not directly instantiate the MessageOfTheDayService, spring will do it instead.
- Client now will have a reference to the MessageOfTheDayService, so it can call the method.

Configuration file

There is a file in the src folder which is called *application.xml*. We add the following code between the <beans> tag:

```
<beans>
      <bean id="msgService"
            class="MessageOfTheDayBasicImpl">
            <property name="message" value="Hello Spring"/>
      </bean>
</beans>
```

Client class

Now we create the *ClientApplication* class as our client code.

In *ClientApplication*, instead of creating *MessageOfTheDayService* directly, we ask the spring container to configure it for us.

In the main method, we create a spring container using *ClassPathXmlApplicationContext* class which gets the *application.xml* as the argument. This class is a container. In java *ApplicationContext* is a container.

```
ClassPathXmlApplicationContext container = new
        ClassPathXmlApplicationContext("application.xml");
```

Through this container, we will get access to the beans that we have configured.
In the configuration file, we mapped the *id: msgService* to the *MessageOfTheDayService*.

So we can access the bean from the container by calling the getBean method:

```
MessageOfTheDayService helloSpring = container.getBean("msgService",
MessageOfTheDayService.class);
```

getBean method takes the name of the bean and the type of the object as the arguments.

application.xml:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
            "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
```

```
      <bean id="msgService"
            class="MessageOfTheDayBasicImpl">

            <property name="message" value="Hello Spring"/>
      </bean>
</beans>
```

And here is the *ClientApplication* class:

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ClientApplication {

    public static void main(String[] args) {
      ClassPathXmlApplicationContext container = new
ClassPathXmlApplicationContext("application.xml");

      MessageOfTheDayService helloSpring = container.getBean("msgService",
MessageOfTheDayService.class);
      System.out.println(helloSpring.getTodaysMessage());
      container.close();
     }
}
```

We print out the message of the day and then we close the container.

Now we run the code. We should see the message.

An Alternative implementation of MessageOfTheDayService

Having the values of the property in the class

The functionality that we get here, is that the message of the day can be changed easily, by a system administrator without recompiling the code.

We just need to go to our xml file, and change the message. We also have the flexibility to change the implementation of the interface in the xml file.

So we can have an alternative implementation of *MessageOfTheDayService* interface with the name: *MessageOfTheDayDynamicImpl*

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MessageOfTheDayDynamicImpl implements MessageOfTheDayService {
	private String[] messages= new String [] {
			"Monday message",
			"Tuesday message",
			"Wednesday message",
			"Thursday message",
			"Friday message",
			"Saturday message",
			"Sunday message",
	};

	@Override
	public String getTodaysMessage() {
		int day = new GregorianCalendar().get(Calendar.DAY_OF_WEEK);
		String message = messages[day-2];
		return message;
	}
}
```

Changes in the applications.xml

And in our application.xml file, we change the implementation to *MessageOfTheDayDynamicImpl.*

application.xml:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
		"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
	<bean id="msgService"
		class="MessageOfTheDayDynamicImpl">
	</bean>
</beans>
```

We remove the *property: message* in the application.xml file, because the messages are now hard coded into our java class and we do not need it anymore. We needed to inject the message into the implementation before.

Do we need to change something in our client code?

No, we do nothing in the client code. And the whole thing can be done without recompiling and rebuilding the whole application.
Now we run the code and see if we get the message for today.

The problem with this implementation is that we were hardcoding the messages in the class.

Third implementation (injecting the values of the property using Spring)

To fix this, we create a third implementation for the message of the day:

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MessageOfTheDayConfigurableImp implements
MessageOfTheDayService {
	private String[] messagesList;

	public void setMessageList(String[] messagesList) {
		this.messagesList = messagesList;
	}
	@Override
	public String getTodaysMessage() {
		int day = new GregorianCalendar().get(Calendar.DAY_OF_WEEK);
		String message = messagesList[day-2];
		return message;
	}
}
```

In this class, we are not configuring the messages, and we will inject the values by using spring. So therefore, we need a method: *setMessageList* in the class.

Changes in the application.xml

In *apllication.xml*, we change the implementation of the interface to this new class. And we need to add the property messageList.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
            "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
      <bean id="msgService"
            class="MessageOfTheDayConfigurableImp">
            <property name="messageList">
                  <list>
                        <value>monday message</value>
                        <value>tuesday message</value>
                        <value>wednesday message</value>
                        <value>thursday message</value>
                        <value>friday message</value>
                        <value>saturday message</value>
                        <value>sunday message</value>
                  </list>
            </property>
      </bean>
</beans>
```

We run the code, it works!

Optimization in Client code

Back to the client code, there is an optimization that we can do:

In this block:

```
MessageOfTheDayService helloSpring = container.getBean("msgService",
MessageOfTheDayService.class);
```

We are asking the container for the object we have configured in the xml file by giving the id of *msgService*.
We can remove the *id* from this *getBean* method. By that we are asking the container for a bean which is implementing the *MessageOfTheDayService*.

Change the above to:

```
MessageOfTheDayService helloSpring =
container.getBean(MessageOfTheDayService.class);
```

If you run it, you see that it still works.

The problem with doing this way is that, if we have two beans with different ids and different implementations of the same interface, we will get an exception indicating that: “expected single matching bean but found 2” when running the code.
We will talk about wiring in Spring in the next chapter.