SPRING XML CONFIGURATION

XML CONFIGURATION INTRODUCTION :

* Lets talk about configuring Spring using XML.
* We are going to take the Sample Application that we developed in the last module and wire up that application with XML using the Spring Framework.

WHY USE XML?

* XML configuration was the first method that was available in Spring and still one of the more popular approaches.
* Somethings are simpler with XML than with Java Configurations.
* There is a separation of concerns that organically happens when configuration code is removed and placed in a separate file.
* We are going to create a file called as application-context.xml
* To wire up our new application.
* Its just a standard that’s associated with Spring , we can use other names for the configuration file.

COPY DEMO :

* In the previous module we developed an application and saw the pain points of having to hardcode our creation of objects statements and such.
* RC on the file bar -> Close All.
* Minimize the project on the Package Explorer.
* RC the project -> Copy
* RC on the Package Explorer window and say Paste.
* Project Name : spring\_sample\_xml
* Click OK.
* We have to make one minor change and that is the advantage of using Maven , Open up the POM.xml , You can see that the ArtifactId is still spring\_sample.
* Change the artifactId to spring\_sample\_xml and Save.
* It is a copy of the previous module project spring\_sample and our libraries are already there for us.
* Click on Application.java and Run as Java Application.
* RC on spring\_sample application developed in the previous module and say Close Project.

APPLICATION CONTEXT :

* XML configuration in Spring begins with a file that we need to create called the applicationContext.
* It really is the root of the application configured in Spring and XML. It doesn’t have to be named the same way , its just a loose standard.
* But it has to be a XML file.
* Its basically a HashMap of objects , and we define the HashMap within our applicationContext.
* The objects we have here within the applicationContext are just name value pairs.
* Although its not the intention of Spring , it can be used as a simple registry , we can lookup the beans registered within the HashMap for our application.
* All our XML configurations begin here , we can have other files that look up and pull in as a references.
* There are some Namespaces put together by our Spring Developers for configuration and validation.
* We will look at adding a namespace to the top of our application that we copied in the previous section.
* Basically we put in an XML snippet at the top of the applicationContext and it knows our bean namespaces.

APPLICATION CONTEXT DEMO :

* Spring STS allows us to add configuration code much easier using their built-in wizards but we will look at how to do It without wizards.
* Go to SRC/MAIN/RESOURCES -> RC -> Spring Bean Configuration File.
* File Name : applicationContext.xml
* Select Next and not Finish.
* Just look at all the XSD namespaces that are available and click Finish.
* We have default namespaces that are declared at the top of the document (REFER SCREENSHOT IN THE FOLDER).
* Xsi:schemaLocation which gives us context sensitive help within our application.
* When we write a “<” within our XML page we get a note from the IDE suggesting some tags or namespaces that we could work with.  
  Note : That the <context> namespace is not present because we haven’t added a particular namespace to the top of the document which we will discuss in the next section.  
  It helps us to write valid and well-formed code.
* Valid in the sense it helps us to write code within any location , lets say suppose I add a <bean></bean> , suppose I add <context></context> it gives me an error saying that I need to import a namespace.
* Suppose we are going to use a wizard.
* RC on SRC/MAIN/RESOURCES and click on New-> File.
* FileName : applicationContext2.xml and click Finish.
* If I want the context help though , Im going to have to type whatever was generated as code when I used the Spring Bean Configuration File.
* If I save the file , it has a source and design tab at the bottom , but with Spring Bean Configuration File , it has Source, namespaces , overview , bean and bean graph tabs at the bottom because the IDE has registered this as a Spring Bean Configuration File and the other one as a normal XML file.

NAMESPACES :

* The snippet of the <bean> namespace definition that resides at the top of the XML file (Look at screenshot in the folder). I believe that the misunderstanding of namespaces is the reason why people don’t use XML namespaces much these days.
* Spring has created these namespaces that simply act as a dictionary for the properties that we can use , create and inject into a bean , lets dive in deeper to look at more of those properties.

XML DECLARATION :

* The definition of a bean in XML looks like :   
  <bean   
  name = “customerService”  
  class = “com.pluralsight.service.CustomerServiceImpl”  
  autowire = “byName”>  
  <property name = “0” ref = “customerRespository” />  
  </bean>
* The above bean is the customerService bean , represents where we want to put our business logic into our application .

BEANS :

* <bean name=”customerRepository” class=”com.pluralsight.repository.HibernateCustomerRepositoryImpl”/>
* The XML configuration is composed of beans and beans are basically entity classes.
* They are just POJO’s that we use inside our application context.  
  Beans can be thought of as replacing the keyword “new”.
* So wherever we are using the keyword “new” in our application like   
  CustomerService service = new CustomerServiceImpl();  
  We can now remove this configuration and place that in an XML file.
* Lastly we always want to define the class and use the interface.
* Why would you want to separate configuration from business logic?  
  We can now change our configuration without needing to recompile our code.  
  We can switch from environments like Dev to Test , We can just use another configuration file and not necessary to recompile pieces of our application. This technique is called as separation of concerns

BEAN DEMO :

* Let us now add a bean definition to our application.
* Open applicationContext.xml
* We want to use the Impl file but refer the interface.
* After the closing of the <beans declaration tag> add the following :   
  <bean name=”customerRepository” class=”com.pluralsight.repository.HibernateCustomerRepositoryImpl”></bean>
* We have created our first bean in XML.

SETTER INJECTION & DEMO:

* Now that we have a bean defined , how do we go about and use that bean?
* We have to use injection.
* There are two types of injection  
  Setter Injection   
  Constructor Injection
* Setter Injection is what it sounds like , using the getters and setters.
* Constructor Injection uses the defined constructors.
* You can use both together also.
* Setter Injection is better for existing code.
* To show an example of setter injection :   
  Open applicationContext.xml  
  Create a new bean :   
  <bean name=”customerService” class=”com.pluralsight.service.CustomerServiceImpl” >  
  <property name=”customerRepository” ref=”customerRepository”></property>  
  </bean>  
  We don’t have our repository and service bean wired together.  
  Open CustomerServiceImpl.java  
  We can find the hardcoded reference to our HibernateCustomerRespositoryImpl  
  Remove the HibernateCustomerRepositoryImpl reference.  
  RC on CustomerRepository customerRepository;  
  Generate getters and setters  
  De-select getCustomerRepository cause we are going to use only setters.  
  Click Ok  
  Source -> Sort Members  
  Remove Unused Imports  
  Go back to applicationContext.xml  
  Add property element to customerService <bean>  
  Everything is wired up correctly and we have done setter injection byName by passing the CustomerRepository class bean name to our CustomerService class reference property
* Go ahead and run the application as RunAs Java Application.
* Before we can run we need to import our applicationContext.xml in our Application.java file.
* Comment out the CustomerService bean
* Type the following in your Application.java  
  ApplicationContext appContext = new ClassPathXmlApplicationContext(“applicationContext”);  
    
  This will look on our classpath for us and find our applciationContext.xml file , Since we are using Maven it will automatically put our file in the right place. Our applciationContext is in SRC/MAIN/RESOURCES -> Maven will put that file into the root of our application.  
    
  We don’t need to put any path information , all we need to do is put the file name as a string parameter.   
    
  Now we have to create a reference to our CustomerService bean  
    
  CustomerService service = appContext.getBean(“customerService”,CustomerService.class);  
    
  Here above if you notice , we are using the interface and not the CustomerServiceImpl class , cause we are swapping it out behind the scenes.
* Now go on and Run the application -> Run As Java Application.  
    
  In the console it shows the following :  
  Our ClassPathXmlApplicationContext has been loaded  
  It has gone and found the bean that we had at the root of the structure  
  It has then started up our application   
  It pulled the resources our of the Bean Definition
* Our application could now have different types of repository injected in and we wouldn’t have to change pieces in our CustomerServiceImpl , this is some of the values of using setter injection and moreover dependency injection.

CONSTRUCTOR INJECTION :

* We have seen how simple setter injection is to use , but what about constructor injection?
* Constructor Injection guarantees us a few nice things using simple constructs in Java.
* We have a defined contract when we define an object.
* The negative is that I need to define a constructor for every situation.
* They can both be used together cause they work well together.
* Slight difference – Index based and not name Based.

CONSTRUCTOR INJECTION DEMO :

* Go to applicationContext.xml
* Change the <property> tag to <constructor> tag.  
  <constructor-arg index=”0”></constructor>  
  Once we save the code , we get an error message that it cant find the constructor.
* Open the CustomerServiceImpl.java
* We can initialize a default no-arg constructor   
  public CustomerServiceImpl() {}
* Now define a constructor with CustomerRepository as parameter.  
  public CustomerServiceImpl(CustomerRepository customerRepository) { this. customerRepository=customerRepository;}
* All I needed to do was change the property tag to a constructor-arg tag in applciationContext.xml and go on to define a constructor in CustomerServiceImpl class with CustomerRepository reference as a parameter.
* Run the Application RunAs – Java Application
* It is index based.
* Gives a more sense of security that we have initialized in the proper way.
* We can use both together , by adding a <property> tag below the <constructor-arg> tag.

AUTOWIRE :

* Early on Spring got a bad reputation for having so many XML Configurations and people didn’t care to go through and wire every bean
* To counter this , then came through with a mechanism to Autowire beans together.
* There are 4 types of Autowiring.  
  byType – This allows a property to be autowired if exactly one bean of that property type exists within the container.  
  For Eg : Our CustomerRepository was of type HibernateCustomerRepositoryImpl  
  byName – byType allows us to create only one instance of that class in the container , whereas byName will allow us to have multiples and choose based on the name.  
  byConstructor – Very similar to setter Injection byType.  
  No or None – It cannot be autowired at all.

AUTOWIRE BYCONSTRUCTOR DEMO :

* Open applicationContext.xml
* We have already defined a <constructor-arg> tag.
* Comment out the <constructor-arg> line
* Come into the element of bean name=”customerService” and add   
    
  autowire=”constructor”
* The application runs well without the <constructor-arg> tag because it automatically autowires our bean named customerService with customerRepository.
* Remove that autowire statement and now run the application.
* It throws a NullPointerException saying that we don’t have a reference to customerRepository from customerService.
* Lets again add the autowire=”constructor”.

AUTOWIRE BYNAME AND BYTYPE DEMO :

* Open up applicationContext.xml
* Go to the autowire statement in the bean and change it from constructor to byType.
* Open up CustomerServiceImpl.java
* We have a default no-arg constructor , A setter .
* For byType it is going to look at the setter and check the parameters whether It is of Type CustomerRepository.
* Run the application and check that it works properly , cause that’s all we have to do for byType.
* Cause it is going to use our setter.
* Go to applicationContext.xml
* Change the autowire line from byType to byName.
* Run the application , it still works.
* If suppose we change the bean name of CustomerService to foo and run the program , it will not work as it wont be able to find a reference by the name of foo in CustomerServiceImpl.
* Go to CustomerServiceImpl.java
* Change the setCustomerRepository() to setFoo()
* Now run the application , by bean naming conventions , it will find the setter and the application runs fine.
* If suppose we change the bean name back from foo to customerRepository and run the application , it will look for a setter by the name of customerRepository ie setCustomerRepository() and the application breaks cause we already changed it to setFoo() in the CustomerServiceImpl.java
* If suppose we switch the autowire=”byName” to autowire=”byType” , it wont care that the name is setFoo() in the CustomerServiceImpl.java
* Change back everything to the way it was.

SUMMARY :

* In this module , we talked about what the applicationContext.xml is and how it’s the root of our application.
* We looked at bean definition.
* Looked at setter injection and how it works with existing code
* Looked at constructor injection.
* Finally we looked at autowiring scenarios.