SPRING CONFIGURATION USING JAVA

JAVA CONFIGURATION INTRODUCTION :

* In this module , we will walk through the configuration of Spring by using Java alone.
* We are going to take the sample application the we created previously and wire that application by using just the Java annotation and the configuration loader.
* It is the latest method available.
* It was brought because people didn’t want to mix XML configuration with the source code.
* We are going to copy our sample project and wire it up with Java Annotations.

COPY DEMO :

* RC on spring\_sample project in the Package Explorer.
* Click on Copy.
* Paste it in the Package Explorer
* Name : spring\_sample\_java -> Ok.
* Go to the pom.xml and change the artifactId to spring\_sample\_java.
* Open the Application.java file and run as java application.

APPCONFIG & DEMO :

* The first thing to note about the Java Configuration is obviously there is no XML or applicationContext.xml
* We had too much XML and they were mocking that Spring developers were more of XML developers and not Java developers.
* Later versions of Spring included namespaces and developers wanted to see less or no use of XML.
* So Java Configuration came into place and everything in Spring can be done with Java Configuration.
* We now create a AppConfig.java file.
* RC on default package -> New -> Class
* Name : AppConfig -> Finish
* Lets look at the annotation that we need to add to make this a configuration file.

@CONFIGURATION & DEMO :

* To replace our XML configuration , we are going to use an @Configuration annotation.
* The java files that have the @Configuration annotation replace any XML configuration file that we may have previously used.
* @Configuration is done at a class level annotation (Look at the screenshot in the folder.)
* Methods that are used with the Bean annotation are defined by @Bean.
* We add a method level annotation for @Bean , it is now registered inside of Spring.
* Classes and method names can be anything. As long as we add the method with @Bean annotation and is within the @Configuration file.
* Go to AppConfig.java
* Above the class add the @Configuration annotation. (org.springframework.context.annotation.Configuration)
* This will tell Spring to go and check for any configurations from this file.
* We are going to create a method   
  @Bean(name = “customerService”) ---- org.springframework.context.annotation.Bean  
  public CustomerService getCustomerService()  
  {  
  return new CustomerServiceImpl();  
  }
* The above is to create a method and annotate it with @Bean because it’s a method level annotation that takes in a class as implementation and returns the Interface.
* We are going to use camel-case and bean notation standards.
* We are going to see how we can use this for setter injection inside our application.

SETTER INJECTION & DEMO :

* Setter injection using the Java Configuration approach is as simple as a method call.
* A lot of the mystery of dependency injection just goes away.
* With java configuration approach it is more transparent as to what is going on.
* Setter injection is simply a matter of calling the setter on a Bean.  
  @Bean(name=”customerService”)  
  public CustomerService getCustomerService()  
  {  
  CustomerServiceImpl customerService=new CustomerServiceImpl();  
  customerService.setCustomerRepository(getCustomerRepository());  
  return customerService();  
  }  
    
  @Bean(name=”customerRepository”)  
  public CustomerRepository getCustomerRepository()  
  {  
  return new HibernateCustomerRepositoryImpl();  
  }
* The above code represents setter injection where both the CustomerService and CustomerRepository are wired together.
* We have a getCustomerRepository() that returns a bean of name : customerRepository which is registered. Then inside the getCustomerService() method we have a setCustomerRepository() method that gets that bean of name : customerRepository that has been registered by the getCustomerRepository() method and it now returns a bean of name : customerService  
  Therefore both of them are wired together and it acts like a method call.
* Notice that both of them are singleton , that they will execute only once and then it is returned from the cache and they are not executed again.
* Since this was a copy of our original Spring project , there are a couple of things we need to do to make sure that it utilizes setter injection.
* Open up CustomerServiceImpl.java
* Change the hardcoded reference of CustomerRepository to be passed in through a setter.
* Generate getters and setters.
* Deselect the getters cause we need only the setter.
* Switch back to the AppConfig.java
* Create a new bean  
  @Bean (name=“customerRepository”)  
  public CustomerRepository getCustomerRepository()  
  {  
  return new HibernateCustomerRepositoryImpl();  
  }
* Before we go any further , remove the hardcoded reference within the getCustomerService method.
* We will say  
  CustomerServiceImpl customerService = new CustomerServiceImpl();  
  customerService.setCustomerRepository(getCustomerRepository());  
  return customerService;
* Save the file.
* We need to tell our application how to look up beans that we have configured in our file.
* Open up Application.java
* Spring wins the award for the longest class name and now we are pointing our application to look at a Configuration class and not at an XML fie.
* Add the following  
  ApplicationContext appContext = new AnnotationConfigApplicationContext(AppConfig.class);  
  CustomerService customerService= appContext.getBean(“customerService” , CustomerService.class);
* Run the code and we have done everything we need to look up our configuration and look up our bean inside of that configuration.

CONSTRUCTOR INJECTION :

* Constructor injection is like setter injection.
* We create our bean instance like how we did before , instead of calling the setter we call the defined constructor that we have for that instance.
* getCustomerRepository() will be used within the constructor.
* One difference being that now we have it stored in the container and we are not passing around objects , we can use bean and getBean aliases which we have defined on our objects.
* To convert it to constructor injection :
* Open up CustomerServiceImpl.java
* Create a constructor   
  public CustomerServiceImpl(CustomerRepository customerRepository)  
  {  
  this.customerRepository=customerRepository;  
  }
* Its going to break our AppConfig when we save it.
* Comment out the service.getCustomerRepository in AppConfig.java
* Within the new CustomerServiceImpl(getCustomerRepository());
* Now this is converted to constructor injection.
* RC and run as java application.
* We don’t need to store instances of these because Spring will store them in the applicationContext for us and when we call it it will be efficient for us.
* It only runs one time and returns the cached instance every other time for us.

AUTOWIRED :

* Since it was only one more line of configuration to use Autowired inside the java configuration project , I decided to include it into this module rather than a whole new module for Autowired specifically..
* To Autowire our application using Java Configuration , We need to add the @ComponentScan annotation , we specify this so that the component scanner knows where it should scan for @Autowired annotation.  
  @ComponentScan({“com.pluralsight”})
* We can use byName or byType ie @Bean or InstanceType.
* One thing that is better with the Java Configuration than the XML Configuration is that using java I can mix pieces that I want. I can have a bean and a definition for the bean and autowire another Bean into it. It makes more sense where these beans are coming from.

AUTOWIRED SETUP DEMO :

* To configure our code to use autowiring :
* In AppConfig.java
* On top of the class write @ComponentScan({“com.pluralsight”}) – Array String syntax
* We will look at couple scenarios where we could be using @Autowired in our code.
* Open up CustomerServiceImpl.java
* Add Syso to the constructor and setter.
* Go on a run the code from Application.java
* We also need to add a default no-arg constructor to CustomerServiceImpl.java  
  public CustomerServiceImpl()  
  {  
  }
* Go back to using setter injection.
* We converted it from using the constructor injection to setter injection.
* Open Application.java and start running it.

AUTOWIRED CONFIGURATION DEMO :

* Now we can see the variations of using Autowired.
* Open up CustomerServiceImpl.java
* Go up above the CustomerRepository and say @Autowired (org.springframework.beans.factory.annotation.Autowired)
* Go to AppConfig.java and comment out the customerService.setCustomerRepository() line.
* We are not going to setCustomerRepository , its going to be Autowired in there.
* Run as Java Application from Application.java
* Now lets use @Autowired on Setter Injection.
* Move it from member level to setter.
* Run as Java Application from Application.java
* In AppConfig.java -> Comment out the Repository Bean all together.
* Open up HibernateCustomerRepositoryImpl.java
* There were somethings from the previous modules called as Stereotype annotations.
* Add @Repository(”customerRepository”)
* Run as Java Application
* Therefore we don’t need the customerRepository bean in the AppConfig.class wherein we have the scanner that will look at our packages and find that we have a bean that is defined for HibernateCustomerRepositoryImpl class as @Repository and inject that inside our code.
* Go back to AppConfig.java
* Comment out CustomerService bean.
* Go to CustomerServiceImpl.java
* We are going to use one of the stereotype annotations called @Service.
* @Service(“customerService”)
* There is now no bean defined within our AppConfig.java
* Everything is done using autowiring.
* So in CustomerServiceImpl we tell it’s a customerService bean , we autowire in the customerRepository.

SUMMARY :

* To recap what we did ,
* Instead of an applicationContext.xml we can have an AppConfig.class as Java Configuration.
* We use an @Configuration to define files that contain the configuration code.
* We can define a bean using @Bean annotation.
* We saw setter and constructor injection.
* Lastly we looked at Autowiring using various approaches and saw the power of autowired.
* Next let us look at some bean scopes.