**INTERNITY FOUNDATION**

**TASK-17**

**Submitted By:**

**Amisha Singhal**

**Java Batch**

**Spring - Beans Auto-Wiring**

**Ans-** In Spring framework, we can wire beans automatically with auto-wiring feature. To enable it, we just define the “**autowire**” attribute in .The Spring container can **autowire** relationships between collaborating beans without using and elements which help cut down on the amount of XML configuration

***Autowiring Modes***

Spring supports the following autowiring modes:

**no:** It’s the default autowiring mode. It means no autowiring.

**byName:** The byName mode injects the object dependency according to name of the bean. In such a case, the property and bean name should be the same. It internally calls the setter method.

**byType:** The byType mode injects the object dependency according to type. So it can have a different property and bean name. It internally calls the setter method.

**constructor:** The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having a large number of parameters.

**autodetect:** In this mode, Spring first tries to autowire by the constructor. If this fails, it tries to autowire by using byType.

**Example-**

We’ll create a simple Java Bean, named Department. Department will have department name property with getter and setter methods. After that, we will initialize this property value in the Spring bean configuration file.

public class Department {

private String deptName;

public String getDeptName() {

return deptName;

}

public void setDeptName(String deptName) {

this.deptName = deptName;

}

}

Created our Employee class, in which we will inject Department bean through Spring autowiring.

public class Employee {

private int eid;

private String ename;

private Department department;

public int getEid() {

return eid;

}

public void setEid(int eid)

this.eid = eid;

}

public String getEname() {

return ename;

}

public void setEname(String ename) {

this.ename = ename;

}

public Department getDepartment() {

return department;

}

public void setDepartment(Department department) {

this.department = department;

}

public void showEployeeDetails(){

System.out.println("Employee Id : " + eid);

System.out.println("Employee Name : " + ename);

System.out.println("Department : " + department.getDeptName());

}

}

Now, looking at the Spring bean configuration file, it is the main part of any Spring application.

<?xml version="1.0" encoding="UTF-8"?>

<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.xsd

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

http://www.springframework.org/schema/context/spring-context.xsd">

<bean id="department" class="guru.springframework.autowiringdemo.Department">

<property name="deptName" value="Information Technology" />

</bean>

<bean id="emp" class="guru.springframework.autowiringdemo.Employee" autowire="byName"></bean>

</beans>

Now, our Spring application is ready with all types of Spring autowiring. So, let’s write a simple test program to see if it works as expected.

@SpringBootApplication

public class AutowiringdemoApplication {

public static void main(String[] args) {

SpringApplication.run(AutowiringdemoApplication.class, args);

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

Employee emp = context.getBean("employee", Employee.class);

emp.setEid(101);

emp.setEname("Spring Framework Guru");

emp.showEployeeDetails();

}

}

**Output-**

**Employee Id: 101**

**Employee Name: Spring Framework Guru**

**Department: Information Technology**

**Process finished with exit code 0**

**Annotation Based Configuration**

**Ans-** In Spring Framework annotation-based configuration instead of using XML for describing the bean wiring, we have the choice to move the bean configuration into component class. It is done by using annotations on the relevant class, method or the field declaration. Before performing XML, injection annotation injection is performed. Therefore, the latter one will override the former configuration for the properties.  
By default,Spring annotation wiring is not turned on in Spring Framework. Therefore, we need to enable it before we can use the Spring annotation-based wiring in the Spring Configuration file.

**Example-**

**Student.java file:**

**package com.example;**

**import org.springframework.beans.factory.annotation.Required;**

**public class Student {**

**private Integer age;**

**private String name;**

**@Required**

**public void setAge(Integer age) {**

**this.age = age;**

**}**

**public Integer getAge() {**

**return age;**

**}**

**@Required**

**public void setName(String name) {**

**this.name = name;**

**}**

**public String getName() {**

**return name;**

**}**

**}**

**MainApp.java file:**

**package com.example;**

**import org.springframework.context.ApplicationContext;**

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

**public class MainApp {**

**public static void main(String[] args) {**

**ApplicationContext context = new**

**ClassPathXmlApplicationContext("Beans.xml");**

**Student student = (Student) context.getBean("student");**

**System.out.println("Name : " + student.getName() );**

**System.out.println("Age : " + student.getAge() );**

**}**

**}**

**The content for the configuration file Beans.xml is as defined:**

**<?xml version = "1.0" encoding = "UTF-8"?>**

**<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-3.0.xsd">**

**<context:annotation-config/>**

**<!-- Definition for student bean -->**

**<bean id = "student" class = "com.example.Student">**

**<property name = "name" value = "Amisha" />**

**<property name = "age" value = "21"/>**

**</bean>**

**</beans>**

**Output:**

**Name: Amisha**

**Age: 21**

**Spring - Java Based Configuration**

**Ans-** Main annotations in Spring Java based configuration are @Configuration and @Bean:

* **@Configuration**– Annotating a class with @Configuration indicates that this class is used as a source of bean definitions. Furthermore, @Configuration classes let inter-bean dependencies be defined by calling other @Bean methods in the same class.
* **@Bean**– The @Bean annotation is used to indicate that a method instantiates, configures, and initializes a new object to be managed by the Spring IoC container. The @Bean annotation plays the same role as the element in the XML configuration. You can use @Bean-annotated methods with any Spring @Component. However, they are most often used with @Configuration beans.

**Example-**

We have created a maven project, and here is the sample pom.xml file and spring dependencies. We have added a new dependency called cglib to support java based configurations:

**<project xmlns="http://maven.apache.org/POM/4.0.0"**

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

**xsi:schemaLocation="http://maven.apache.org/POM/4.0.0**

**http://maven.apache.org/xsd/maven-4.0.0.xsd">**

**<modelVersion>4.0.0</modelVersion>**

**<groupId>SpringJavaBasedConfig</groupId>**

**<artifactId>SpringJavaBasedConfig</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**<properties>**

**<spring.version>3.2.0.RELEASE</spring.version>**

**</properties>**

**<dependencies>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-core</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<dependency>**

**<groupId>cglib</groupId>**

**<artifactId>cglib</artifactId>**

**<version>3.0</version>**

**</dependency>**

**</dependencies>**

**</project><div class="open\_grepper\_editor" title="Edit & Save To Grepper"></div>**

**Basic interface:**

**package com.java2novice.bean;**

**public interface MyColor {**

**public void printColor();**

**}**

**<div class="open\_grepper\_editor" title="Edit & Save To Grepper"></div>**

**Implementation class:**

**package com.java2novice.bean;**

**public class RedColor implements MyColor{**

**@Override**

**public void printColor() {**

**System.out.println("It is red in color...");**

**}**

}

<div class="open\_grepper\_editor" title="Edit & Save To Grepper"></div>

Now here comes the java based configuration file. This class is equivalent of xml based configuration file.The bean declaration can be achieved by using @Bean annotation.

**package com.java2novice.config;**

**import org.springframework.context.annotation.Bean;**

**import org.springframework.context.annotation.Configuration;**

**import com.java2novice.bean.MyColor;**

**import com.java2novice.bean.RedColor;**

**@Configuration**

**public class MyAppConfig {**

**@Bean(name="myColorBean")**

**public MyColor getMyColors(){**

**return new RedColor();**

**}**

**}**

**<div class="open\_grepper\_editor" title="Edit & Save To Grepper"></div>**

**Project structure snapshot:**

Spring application structure

**Demo class:**

**package com.Amisha;**

**import org.springframework.context.ApplicationContext;**

**import org.springframework.context.annotation.AnnotationConfigApplicationContext;**

**import com.java2novice.bean.MyColor;**

**import com.java2novice.config.MyAppConfig;**

**public class SpringDemo {**

**public static void main(String a[]){**

**ApplicationContext context**

**= new AnnotationConfigApplicationContext(MyAppConfig.class);**

**MyColor color = (MyColor) context.getBean("myColorBean");**

**color.printColor();**

**}**

**}**

**<div class="open\_grepper\_editor" title="Edit & Save To Grepper"></div>**

**Output:**

**It is red in color...**