**IOC(Inversion of Control)**

**Inversion of Control is a design principle where control is being transfer to the container for the creation of the object and not only for the creation of object but also for the flow of application. In case of Spring there is IOC Container where object is created. Now we have created the object, and the question arises how we are going to use this Object then one more design pattern comes into picture i.e.**

**DI(Dependency Injection)**

**Now suppose we have two class i.e.**

**Class CPU**

**{**

**}**

**Class Laptop{**

**}**

**We need the object of CPU inside the laptop to work, object is created inside the IOC container now how we are going to inject CPU object in Laptop class , her comes the Dependency Injection which inject the the object of CPU in the laptop class.**

**IOC is a design principle and Dependency Injection is a design pattern.**

**Spring VS Spring Boot**

|  |  |
| --- | --- |
| **Spring** | **Spring Boot** |
| **Spring is an open-source lightweight framework widely used to develop enterprise applications.** | **Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.** |
| **The most important feature of the Spring Framework is dependency injection.** | **The most important feature of the Spring Boot is Autoconfiguration.** |
| **It helps to create a loosely coupled application.** | **It helps to create a stand-alone application.** |
| **To run the Spring application, we need to set the server explicitly.** | **Spring Boot provides embedded servers such as Tomcat and Jetty etc.** |
| **To run the Spring application, a deployment descriptor is required.** | **There is no requirement for a deployment descriptor.** |
| **To create a Spring application, the developers write lots of code.** | **It reduces the lines of code.** |
| **It doesn’t provide support for the in-memory database.** | **It provides support for the in-memory database such as H2.** |
| **Developers need to write boilerplate code for smaller tasks.** | **In Spring Boot, there is reduction in boilerplate code.** |
| **Developers have to define dependencies manually in the pom.xml file.** | **pom.xml file internally handles the required dependencies.** |

**Let’s create a spring boot application and then we will move towards to spring boot:-**

**First we will create simple java class i.e. Alien.class**

**public class Alien{**

**public void code(){**

**System.out.println(“coding”);**

**}**

**}**

**Now we have create the object inside the container of the JVM and we can do through ApplicationContext, Let’s go to the main application and see the method.**

**SpringApplication.run(SpringBootFirstApplication.class, args);**

**If we go the implementation of the run method, then we will find that the return type of run method is ConfigurableApplicationContext and this ConfigurableApplicationContext return ApplicationContext.**

**So what we will do is :-**

**ApplicationContext context= SpringApplication.run(SpringBootFirstApplication.class, args);**

**Alien obj=context.getBean(Alien.class);**

**obj.code();**

**Let’s see the whole code:-**

**@SpringBootApplication**

**public class SpringBootFirstApplication {**

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

**ApplicationContext context=SpringApplication.run(SpringBootFirstApplication.class, args);**

**Alien obj=context.getBean(Alien.class);**

**obj.code();**

**}**

**}**

**After communicating with the IOC Container it still don’t create the object of the Alien class , then what is the problem now before creating the object of any class spring always check if spring need to create the object of that class or not it check by verifying the annotation i.e. @Component**

**So we need to add @Component annotation above the Alien class to signify that we need to create the object of that class.**

**@Component**

**public class Alien {**

**public void code() {**

**System.out.println("coding ");**

**}**

**}**

**Now let’s add another layer in the application , now suppose in alien class Object of Laptop class is calling the method Compile method and in that compile method it is doing the work.**

**@Component**

**public class Alien {**

**Laptop laptop;**

**public void code() {**

**laptop.compile();**

**}**

**}**

**@Component**

**public class Laptop {**

**public void compile() {**

**System.out.println("compiling......");**

**}**

**}**

**In this way, if we execute the program from the main application it will throw an error as in application context We are calling Alien class and here laptop class is not wired properly with the Alien class.**

**What we need to do is we need to create the connection between the laptop class and alien class and we do with the help of one annotation i.e. @Autowired above the laptop object.**

**@Component**

**public class Alien {**

**@Autowired**

**Laptop laptop;**

**public void code() {**

**laptop.compile();**

**}**

**}**

**@Component**

**public class Laptop {**

**public void compile() {**

**System.out.println("compiling......");**

**}**

**}**

**@SpringBootApplication**

**public class SpringBootFirstApplication {**

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

**ApplicationContext context=SpringApplication.run(SpringBootFirstApplication.class, args);**

**Alien obj=context.getBean(Alien.class);**

**obj.code();**

**}**

**}**

**The above code is proper application for the spring boot.**

**Let’s start the spring framework to understand the behind the scene of the spring framework.:-**

**First we will create one project using Maven where we need to create one simple application like above suppose we need to create a alien class where coding method is there which is printing coding:-**

**public class App**

**{**

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

**{**

**Alien obj = new Alien();**

**obj.code();**

**}**

**}**

**Like above and we want to create the object of that class with the help of spring boot to do that we need to create IOC container where object are kept let’s see how to create the application context in the spring:-**

**To create the application context we need to add some dependencies in the maven project by adding dependency in the .pom file.**

**After that we need to create the container or we can say we need to connect with the container in the spring and we can do so by using application context:-**

**ApplicationContext context = new ClassPathXmlApplicationContext();**

**Alien obj =(Alien) context.getBean("alien");**

**By using this line we can connect to the IOC Container:-**

**Now if we run it by using the main application it will still show the error: as we have not specified the class which spring need to create the object. We can do so by annotation and XML, Let’s see how to do with the xml form:-**

**We need to create the package with name resource inside the main package and inside the resource package we need to create the spring.xml file like this:-**

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

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

**xsi:schemaLocation="**

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

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

**</bean>**

**</beans>**

**Main application will be**

**public class App**

**{**

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

**{**

**ApplicationContext context = new ClassPathXmlApplicationContext("resources\\spring.xml"); //Specify the sprin.xml**

**Alien obj =(Alien) context.getBean("alien"); //id name in xml specified here**

**obj.code();**

**}**

**}**

**Bean class is here**

**public class Alien {**

**public void code() {**

**System.out.println("Coding");**

**}**

**}**

**Object Creation:-**

**Object is created once it is specified in xml, we don’ t need to call the object by using application context. If we specify two bean for the same class it will create the object two times for the same class.**

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

**</bean>**

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

**</bean>**

**But we try to create bean from the main application by using getBean with the help of application context it will create only one object as by default scope of the bean is singleton**

**Alien obj =(Alien) context.getBean("alien");**

**obj1.code();**

**Alien obj1 =(Alien) context.getBean("alien");**

**obj1.code();**

**Scope:-**

**In case of Spring core there are two scope present :-**

1. **Singleton**
2. **Prototype Scope**

**By default we have singleton Scope let’s understand with the example:-**

**Let’s suppose we have AGE variable in the alien class and we are trying to access the age variable from the main application with two getBean Like this:-**

**public class Alien {**

**int age;**

**public void code() {**

**System.out.println("Coding");**

**}**

**}**

**Main application class:-**

**public class App**

**{**

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

**{**

**ApplicationContext context = new ClassPathXmlApplicationContext("resources\\spring.xml");**

**Alien obj =(Alien) context.getBean("alien");**

**obj.age=21;**

**System.out.println(obj.age);**

**Alien obj2 =(Alien) context.getBean("alien");**

**System.out.println(obj2.age);**

**}**

**}**

**Output=**

**21**

**21**

**Here output will be same as by default it is singleton scope so when we try to change the value of age with reference it will get reflected in the other object also now suppose we need to change the scope of the bean then we need to change the in xml in this way**

**<bean id="alien" class="com.Telusko.Spring1.Alien" scope="prototype">**

**</bean>**

**Now after changing the scope in the xml file and if we try to execute main application by using woe getbean it will show different result:-**

**Output=**

**21**

**0**

**So whenever we change the scope in xml file and try to access the value of variable with two different object by using getBean it will create new object every time and value of the variable get change.**

**PrototypeScope:- In this scope whenever we try to access the bean using getBean it will create the new object and supply it to the application. In case of prototype object is not created automatically as we need to call getbean to create the object.**

**But in case of singleton object is created automatically.**

**Setter Injection**

**Now we have seen earlier how to inject the value of age variable , we have injected the value earlier by using the object now we try to inject the value using setter to do so let’s see what need to be done.**

**First we need to create the setter and getter of the age variable as shown below:-**

**public class Alien {**

**private int age;**

**public void code() {**

**System.out.println("Coding");**

**}**

**public int getAge() {**

**return age;**

**}**

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

**this.age = age;**

**}**

**}**

**Now what we need to do some changes in xml by using property let’s see how:-**

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

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

**</bean>**

**Here property name is age which is same as variable name of the age. It set the value of age using setter.**

**Main Application java**

**public class App**

**{**

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

**{**

**ApplicationContext context = new ClassPathXmlApplicationContext("resources\\spring.xml");**

**Alien obj =(Alien) context.getBean("alien");**

**obj.setAge(21);**

**System.out.println(obj.getAge());**

**//obj.code();**

**Alien obj2 =(Alien) context.getBean("alien");**

**System.out.println(obj2.getAge());**

**//obj2.code();**

**}**

**}**

**Now if we execute this application we will set the variable age using setter method of the alien class.**

**Reference Attribute**

**Now we have understood how to inject the variable using property till now we have injected only primitive value using setter now what if the value is reference variable, let’s see how we will do.**

**Let’s understand the application , we have Alien class where we have age variable and Laptop reference let’s see :-**

**public class Alien {**

**private int age;**

**private Laptop lap;**

**public void code() {**

**System.out.println("Coding");**

**lap.compile();**

**}**

**public Laptop getLap() {**

**return lap;**

**}**

**public void setLap(Laptop lap) {**

**this.lap = lap;**

**}**

**public int getAge() {**

**return age;**

**}**

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

**this.age = age;**

**}**

**}**

**Using lap reference we are calling the compile method of the laptop in the code method. Let’s see the laptop class what is written there:-**

**public class Laptop {**

**public Laptop() {**

**System.out.println("Laptop object created");**

**}**

**public void compile() {**

**System.out.println("compiling");**

**}**

**}**

**Now let’s see the main application program in the spring:-**

**public class App**

**{**

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

**{**

**ApplicationContext context = new ClassPathXmlApplicationContext("resources\\spring.xml");**

**Alien obj =(Alien) context.getBean("alien");**

**obj.setAge(21);**

**System.out.println(obj.getAge());**

**obj.code();**

**}**

**}**

**Now let’s see the xml file how to do the mapping:-**

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

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

**<property name="lap" ref="lap1"></property>**

**</bean>**

**<bean id="lap1" class="com.Telusko.Spring1.Laptop">**

**</bean>**

**To set the reference value in the xml we will use reference variable and inside ref we will pass the id name of the laptop class.**

**Constructor Injection**

**Earlier we have seen setter injection now suppose if we want to inject the value of age using constructor in the alien class let’s see what need to be done in the xml file:-**

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

**<constructor-arg value="21"/>**

**<property name="lap" ref="lap1"/>**

**</bean>**

**In the above example we have injected one parameter inside the constructor now what if we want to inject two parameter inside the constructor one will be reference and other will be age , now see how we are going to do it:-**

**Let’s see the code first:-**

**public Alien(int age, Laptop lap) {**

**super();**

**this.age = age;**

**this.lap = lap;**

**}**

**The Equivalent code for the xml will be:-**

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

**<constructor-arg value="21"/>**

**<constructor-arg ref="lap"/>**

**</bean>**

**<bean id="lap1" class="com.Telusko.Spring1.Laptop">**

**</bean>**

**Now here the question arise how they are finding the value to injected in the constructor even if are not giving the name , it checks the sequence of the parameter constructor. If we change the sequence of the parameter in xml it will through an error. So if we change sequence it will through an error. To resolve this issue we will add the type in xml.**

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

**<constructor-arg type="com.Telusko.Spring1.Alien" value="21"/>**

**<constructor-arg type="com.Telusko.Spring1.Laptop" ref="lap1"/>**

**</bean>**

Now suppose we have two variable with same class and we have written in different sequence, what needs to be done to resolve the conflict. At that time we are going to use index no. to specify the sequence of the parameter.

**public Alien(int age,int salary,Laptop lap) {**

**super();**

**this.age = age;**

**this.lap = lap;**

**this.salary=salary;**

**}**

And it’s equivalent xml file is written below:-

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

**<constructor-arg index="0" value="21"/>**

**<constructor-arg index="1" ref="lap1"/>**

**<constructor-arg index="2" value="5000"/>**

**</bean>**

**<bean id="lap1" class="com.Telusko.Spring1.Laptop">**

**</bean>**

**AutoWiring by name**

Now suppose we have created a reference of the class with name **com** in the alien and have created the bean id with the same name in xml and want to inject that reference in the object automatically at that that time Autowiring **byname** is used. Let’s see how:-

Alien class:-

**public class Alien {**

**private int age;**

**private Computer com;**

**private int salary;**

**public void code() {**

**System.out.println("Coding");**

**com.compile();**

**}**

**}**

Computer interface

**public interface Computer {**

**void compile();**

**}**

Laptop is injecting computer interface:-

**public class Laptop implements Computer {**

**@Override**

**public void compile() {**

**System.out.println("compiling through laptop");**

**}**

**}**

Now this is the xml file

**<bean id="alien" class="com.Telusko.Spring1.Alien">**

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

**<property name="com" ref="com "/>**

**</bean>**

**<bean id="com" class="com.Telusko.Spring1.Laptop">**

**</bean>**

**\*\*\*here property name is reference variable name and ref in bean id name**

Here property name and bean id is same and we want to autowire it by name what need to be done is to add **autowire =byname** in the bean of alien like this:-

**<bean id="alien" class="com.Telusko.Spring1.Alien" autowire=”byname”>**

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

**</bean>**

**<bean id="com" class="com.Telusko.Spring1.Laptop">**

**</bean>**

It will search the name of the reference which is specified in the class with the name of the bean id and inject it in the object.

**Autowire by type:**

In this if we specify the autowire =”byType” it will search for the type of the reference instead of the name

**<bean id="alien" class="com.Telusko.Spring1.Alien" autowire="byType">**

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

**</bean>**

**<bean id="com" class="com.Telusko.Spring1.Laptop">**

**</bean>**

**<bean id="com" class="com.Telusko.Spring1.Desktop ">**

**</bean>**

**</beans>**

if we comment one bean then it work properly if both the bean is present then it will through an error **autowire=byType** at that time we need need to use autotype=”byname”.

Instead of using autotype=”byname” we can specify one of the bean as **primary. Let’s see how:-**

**<bean id="alien" class="com.Telusko.Spring1.Alien" autowire="byType">**

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

**</bean>**

**<bean id="com1" class="com.Telusko.Spring1.Laptop" primary="true">**

**</bean>**

**<bean id="com2" class="com.Telusko.Spring1.Desktop">**

**</bean>**

**</beans>**

**Lazy Inintialization of bean**

Now some time we don’t want to initialize the bean when the application is boot up , object should be initialize when we need it , it will save the memory. To do so we simply need to add lazy –initialisation=true in the xml like this as shown below:-

**<bean id="com2" class="com.Telusko.Spring1.Desktop" lazy-init="true">**

**</bean>**

**Java Based Configuration**

Till now whatever we have done the configuration is done with the help of xml file but now we are going to try it out using java based Configuration file to manage the java application.

So let’s see what needs to be done for java Based Configuration in main application file , so first of all we need to change the container of the application context inorder to use java based configuration, let’s see how:-

**public class App**

**{**

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

**{**

**ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);**

**Desktop dt =context.getBean(Desktop.class);**

**dt.compile();**

**}**

**}**

What we have done is simple we have change the container i.e.

new AnnotationConfigApplicationContext(AppConfig.class);

Here we are calling the method of Desktop class, now let’s see how we are going to implement the java based configuration:-

🡺First of all we will create an java class and mark it with annotation **@Configuration** as described below and now we will create the method of with return type of the bean class and inside the method we will return the object of the class that our main application needs and marks that method with the annotation **@Bean**

**@Configuration**

**public class AppConfig {**

**@Bean**

**public Desktop desktop() {**

**return new Desktop();**

**};**

**}**

Now we will execute the main application and we are the getting the method that we want to execute in the main application.

**Bean Name:-**

The default name of the bean is the method name that we have specified in the java based Configuration. So if we want to call the bean name in the main application we just need need to provide the method name like this:-

**public class App**

**{**

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

**{**

**ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);**

**Desktop dt =context.getBean(“desktop”,Desktop.class);**

**dt.compile();**

**}**

**}**

if we want to change the name of the bean we just need to add the attribute with the **@Bean(name=”com”)** in the java based Configuration and we can easily change the name.

**@Configuration**

**public class AppConfig {**

**@Bean(name=”com”)**

**public Desktop desktop() {**

**return new Desktop();**

**};**

**}**

**Scope:-**

We can change the scope of the bean by just adding the one annotation in java based Configuration i.e. **@Scope(“Prototype)**  by default the scope of the bean is singleton but if we change the scope then when we want the object it will create a new scope:-

**@Configuration**

**public class AppConfig {**

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

**@Scope("prototype")**

**public Desktop desktop() {**

**return new Desktop();**

**};**

**}**

**Autowire in Java Based Configuration(Dependency injection)**

Now what we are going to do here is to use alien class for the autowire in the alien class we have computer reference which is dependent of computer class let’ see how we need to do the depency injection in the java based configuration in the main app.

Let’s see the main application here:-

**public class App**

**{**

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

**{**

**ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);**

**Alien obj1=context.getBean(Alien.class);**

**obj1.code();**

**Desktop dt =context.getBean("com",Desktop.class);**

**dt.compile();**

**}}**

Here we creating the container to create the object of the **Alien** class. Let’s see the structure of Alien class first then we will see how to create the bean of the alien class:-

**public class Alien {**

**private int age;**

**private Computer com;**

**private int salary;**

**public void code() {**

**System.out.println("Coding");**

**com.compile();**

**}**

**public Computer getCom() {**

**return com;**

**}**

**public void setCom(Computer com) {**

**this.com = com;**

**}**

**public int getSalary() {**

**return salary;**

**}**

**public void setSalary(int salary) {**

**this.salary = salary;**

**}**

**public int getAge() {**

**return age;**

**}**

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

**this.age = age;**

**}**

**}**

Now let’s see how we have injected the dependency in the application:-

@Configuration

**public class AppConfig {**

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

**//@Scope("prototype")**

**public Desktop desktop() {**

**return new Desktop();**

**};**

**@Bean**

**public Alien alien(@Autowired Computer com) {**

**Alien obj = new Alien();**

**obj.setAge(21);**

**obj.setCom(com);**

**return obj;**

**};**

**}**

As we can see, we have created the alien method with return type of Alien and there is parameter of Computer class which acts as **autowiring by type** If there is any bean available in the configuration file with type computer it will automatically inject the reference into the method.

With the help of that reference we inject it in the object of the alien.

**Now suppose we have one more bean which need to be injected as computer class implements laptop and desktop class , let’s see how to handle it properly.**

At that **Primary** and **Qualifier is** used to the problem.

Earlier if we see how to handle multiple bean in xml we have two have

🡺First we use **ref** to explicitly define the bean name in the xml

🡺Second we use **primary** to define if two bean are available then this bean is primary.

In the same way in Java base configuration:-

For **ref** we have **@Qualifier** and for primary **@primary**

**@Qualifier** is used with the target bean.

**@primary** is used with usable bean let’ s see how:-

**@Qualifier Annotation**

**@Configuration**

**public class AppConfig {**

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

**//@Scope("prototype")**

**public Desktop desktop() {**

**return new Desktop();**

**};**

**@Bean**

**public Alien alien(@Qualifier("laptop")Computer com) {**

**Alien obj = new Alien();**

**obj.setAge(21);**

**obj.setCom(com);**

**return obj;**

**};**

**@Bean**

**public Laptop laptop()**

**{**

**return new Laptop();**

**};**

**}**

**@primary annotation**

**@Configuration**

**public class AppConfig {**

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

**//@Scope("prototype")**

**public Desktop desktop() {**

**return new Desktop();**

**};**

**@Bean**

**public Alien alien(Computer com) {**

**Alien obj = new Alien();**

**obj.setAge(21);**

**obj.setCom(com);**

**return obj;**

**};**

**@Bean**

**@Primary // we just need to put this annotation above the usable bean to make it primary.**

**public Laptop laptop()**

**{**

**return new Laptop();**

**};**

**}**

**Component Sterotype Annotation:-**

**@Component**

If we need the object of class automatically from the spring we just need to add **@Component annotation** above the class it simply make the object without defining the it in the xml or java based configuration.

Let’s see how from the beginning:-

**public class App**

**{**

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

**{**

**ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);**

**Alien obj1=context.getBean(Alien.class);**

**obj1.code();**

**Desktop dt =context.getBean("com",Desktop.class);**

**dt.compile();**

**}**

**}**

As we can see here we are creating the container by using **AnnotationConfigApplicationContext(AppConfig.class)** so we need to create the empty class **AppConfig.class** annoted with **@Component** and **@ComponentScan** Component scan all the object which needs to created like this:-

@Configuration

@ComponentScan("com.Telusko.Spring1")

public class AppConfig {

}

Now let’s see the all target and usable class here:-

**@Component**

**public class Desktop implements Computer{**

**public Desktop() {**

**System.out.println("Desktop object created");**

**}**

**@Override**

**public void compile() {**

**// TODO Auto-generated method stub**

**System.out.println("Compiling through desktop");**

**}**

**}**

**Alien class:-**

**@Component**

**public class Alien {**

**private int age;**

**private Computer com;**

**private int salary;**

**public void code() {**

**System.out.println("Coding");**

**com.compile();**

**}**

**public Computer getCom() {**

**return com;**

**}**

**public void setCom(Computer com) {**

**this.com = com;**

**}**

**public int getSalary() {**

**return salary;**

**}**

**public void setSalary(int salary) {**

**this.salary = salary;**

**}**

**public int getAge() {**

**return age;**

**}**

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

**this.age = age;**

**}**

**}**

**Computer class**

**@Component**

**public interface Computer {**

**void compile();**

**}**

**Laptop class**

**@Component**

**public class Laptop implements Computer {**

**public Laptop() {**

**System.out.println("Laptop object created");**

**}**

**@Override**

**public void compile() {**

**System.out.println("compiling through laptop");**

**}**

**}**

From this we only create the object, we haven’t inject the object in the application:-

**Desktop object created**

**Laptop object created**

**Coding**

Let’s see how to inject the dependency injection in the application next:-

@Component

public class Alien {

private int age;

@Autowired

@Qualifier("desktop")

private Computer com;

private int salary;

public void code() {

System.out.println("Coding");

com.compile();

}

public Computer getCom() {

return com;

}

public void setCom(Computer com) {

this.com = com;

}

public int getSalary() {

return salary;

}

public void setSalary(int salary) {

this.salary = salary;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

We just need to add add **@Autowired** in the reference of the computer and to specify explicitly which bean we need to use by **@Qualifier(“laptop”)** to specify the bean name . The default bean name of the class is class name but start the class name with the small case alphabet.

**@Primary Annotation**

Primary annotation is generally used to specify the bean class which needs to be primarily injected as dependency injection like this:-

**@Component**

**@Primary**

**@Scope(“prototype”)**

**public class Desktop implements Computer{**

**public Desktop() {**

**System.out.println("Desktop object created");**

**}**

**@Override**

**public void compile() {**

**// TODO Auto-generated method stub**

**System.out.println("Compiling through desktop");**

**}**

**}**

This primary annotation makes the desktop class primary which will be injected automatically if it is **@AutoWired**

**SpringBoot**

Whatever we have done in the Spring Core, we will do the same thing in the Spring boot let’s see how:-

What will be the archietecture of Springboot application:-

🡺First of all , We will have SpringBootFirstApplication in which we will contain the application Container for the **Alien Class** annotated with **@Component**

**🡺**Now will create the Interface **Computer** which has **compile method** in it, this Computer class is implemented by two other class i.e. **Laptop** and **Desktop** annotated with **@Component class,** Both the class acts as a bean

**🡺**Now in the alien class we will have **computer object,** and age variable and then we will create the setter and getter of the two reference and age variable.

🡺 so we need to inject the **Computer object** so we need to use **@Autowired** annotation above the computer object and use the **@Qualifier** annotation to specify which bean to injected as there are two beans here.

🡺Let’s see the code here:-

**SpringBootFirstApplication**

**@SpringBootApplication**

**public class SpringBootFirstApplication {**

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

**ApplicationContext context=SpringApplication.run(SpringBootFirstApplication.class, args);**

**Alien obj=context.getBean(Alien.class);**

**System.out.println(obj.getAge());**

**obj.code();**

**}**

**}**

**Alien class:-**

**@Component**

**public class Alien {**

**@Value("25")**

**private int age;**

**private Computer com;**

**public Computer getCom() {**

**return com;**

**}**

**@Autowired**

**@Qualifier("laptop")**

**public void setCom(Computer com) {**

**this.com = com;**

**}**

**public void code() {**

**com.compile();**

**}**

**public int getAge() {**

**return age;**

**}**

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

**this.age = age;**

**}**

**}**

**Computer Interface**

**public interface Computer {**

**void compile();**

**}**

**Laptop class**

**@Component**

**public class Laptop implements Computer {**

**public void compile() {**

**System.out.println("Compiling in Laptop....");**

**}**

**}**

**Desktop class**

**@Component**

**@Primary**

**public class Desktop implements Computer {**

**public void compile() {**

**System.out.println("Compiling in Desktop....");**

**}**

**}**

**Different Layers of Spring Boot**

Repository

Service

Controller

In spring boot we have three layer:-

🡺Controller

🡺Service

🡺Repository

🡺Controller

In Spring Boot, we have controller layer which redirect the flow of execution to the desired location using dispatcher servlet.

🡺Service

In Spring Boot, there is Service layer which represents business logic of the application. This class is annotated with **@ServiceLayer**

🡺Repository

In Spring boot, there is repository layer which is DAO layer which helps in connecting the database of using the repository.

**Spring Boot JDBC:-**

To create the Spring Boot JDBC in the spring boot we need to add two dependency in the spring boot project like:-

1. JDBC API

2.My SQL driver

Then we are going to add the properties in the application.properties file of the Spring JDBC like described below:-

**spring.datasource.url=jdbc:mysql://localhost:3306/ineuron**

**spring.datasource.username=root**

**spring.datasource.password=Gaurav45**

**spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver**

Let’s understand with the example:-

MainApplication

(SpringBootapplication

Repository

Student reference

JdbcTemplate

Save()

Findall()

Model

(Student.class)

Name

Rollno

age

Controller

(StudentService)

addStudent()

getStudent()

Second it will call Firstly bean creation student

the controller Property added

to addStudent

At Last it will call

The repository to

Save inside the

Database

Let’s see the code of the model class i.e. Student.java

package com.telusko.SpringJDBCEx.model;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope("prototype")

public class Student {

private int rollNo;

private String name;

private int marks;

@Override

public String toString() {

return "Student [rollNo=" + rollNo + ", name=" + name + ", marks=" + marks + "]";

}

public int getRollNo() {

return rollNo;

}

public void setRollNo(int rollNo) {

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getMarks() {

return marks;

}

public void setMarks(int marks) {

this.marks = marks;

}

}

Now let’s see the controller code here:-

package com.telusko.SpringJDBCEx.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.telusko.SpringJDBCEx.dao.StudentRepo;

import com.telusko.SpringJDBCEx.model.Student;

@Service

public class StudentService {

private StudentRepo repo;

public void addStudent(Student s) {

// TODO Auto-generated method stub

repo.save(s);

}

public StudentRepo getRepo() {

return repo;

}

@Autowired

public void setRepo(StudentRepo repo) {

this.repo = repo;

}

public List<Student> getStudents() {

// TODO Auto-generated method stub

return repo.findAll();

}

}

Now let’s see the main Application code

package com.telusko.SpringJDBCEx;

import java.util.List;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import com.telusko.SpringJDBCEx.model.Student;

import com.telusko.SpringJDBCEx.service.StudentService;

@SpringBootApplication

public class SpringBootJdbcApplication {

public static void main(String[] args) {

ApplicationContext context=SpringApplication.run(SpringBootJdbcApplication.class, args);

Student s=context.getBean(Student.class);

s.setRollNo(105);

s.setName("naveen");

s.setMarks(55);

StudentService service= context.getBean(StudentService.class);

service.addStudent(s);

List<Student> students = service.getStudents();

System.out.println(students);

}

}

Now let’s see the repository code here:-

package com.telusko.SpringJDBCEx.dao;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.ArrayList;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.stereotype.Repository;

import com.telusko.SpringJDBCEx.model.Student;

@Repository

public class StudentRepo {

private Student student;

private JdbcTemplate jdbc;

public JdbcTemplate getJdbc() {

return jdbc;

}

@Autowired

public void setJdbc(JdbcTemplate jdbc) {

this.jdbc = jdbc;

}

public void save(Student s) {

// TODO Auto-generated method stub

String sql= "insert into student (rollno,name,marks) values(?,?,?)";

int rows=this.jdbc.update(sql,s.getRollNo(),s.getName(),s.getMarks());

System.out.println(rows+"effected");

}

public List<Student> findAll() {

// TODO Auto-generated method stub

String sql = "select \* from student";

RowMapper <Student> mapper = new RowMapper<Student>() {

@Override

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

// TODO Auto-generated method stub

Student s = new Student();

s.setRollNo(rs.getInt("rollno"));

s.setName(rs.getString("name"));

s.setMarks(rs.getInt("marks"));

return s;

}

};

return this.jdbc.query(sql, mapper);

}

public Student getStudent() {

return student;

}

public void setStudent(Student student) {

this.student = student;

}

}

**JDBCTemplate**

JDBCTemplate is used in repository class of the springboot application, first we need to create the reference of JDBCTemplate and add annotation **@autoWired.** Springboot look after the dependency injection facility while we create JDBCTEmplate.

**ROWMapper**

RowMapper is generally used to fetch the value from the sql. It is a kind of functional interface which take two parameter

1.Resultset

2.RowNum

We need to pass the resultset of those property which we need to fetch and pass into rowmapper reference. It will give back all the value as described above:-

RowMapper <Student> mapper = new RowMapper<Student>() {

@Override

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

// TODO Auto-generated method stub

Student s = new Student();

s.setRollNo(rs.getInt("rollno"));

s.setName(rs.getString("name"));

s.setMarks(rs.getInt("marks"));

return s;

}

};

**SpringBootWeb**

To create a SpringBootWeb we need to add one Dependency i.e. SpringBoot Web it contains dependency like **spring boot strarter web.** if we create the spring boot web project it contain embedded tomcat server so we don’t need to add external server in it.



Let’s first see file configuration of the spring boot, so in Spring boot there is **SRC folder** inside it there is bydefault **java & resources folder**

Inside java folder we have all java file related to spring boot

Inside resource folder there is application.properties which is responsible for adding some properties which is necessary for spring boot to work.

Now suppose if you want to add some view technology like jsp in spring then we need to create the folder inseide **SRC** with name **webapp** and that JSP page inside it.

**Let’s see the working of the spring boot application:-**

🡺As we know inside spring boot, there is embedded server and if want to create webpage and want to see in web browser what need to be done let’ s see..

View of the page

Controller

Mapped with **@Controller**

Method mapped with **urlpattern** return viewtechnology name(jsp)

Client gives the request using URL

**Convert the jsp by Tom cat jasper dependency into servlet and sent back as reponse**

🡺First of all we need to create the controller which takes the client request so before moving forward for creation of the controller we need to know few annotation which we have used in our controller

* **@Controller**
* **@RequestMapping**

**@Controller:** Indicates that the class is a Spring MVC controller.

**@GetMapping("/"):** Maps HTTP GET requests to the home method.

**@Controller**

**public class HomeController {**

**@GetMapping("/")**

**public String home() {**

**System.out.println("home called");**

**return "index.jsp";**

**}**

**}**

Now let’s see the jsp page:-

**<%@ page language="java" contentType="text/html; charset=UTF-8"**

**pageEncoding="UTF-8"%>**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<meta charset="UTF-8">**

**<title>home</title>**

**</head>**

**<body>**

**<h2> home page</h2>**

**</body>**

**</html>**

Now whenever client request below url to their browser:-

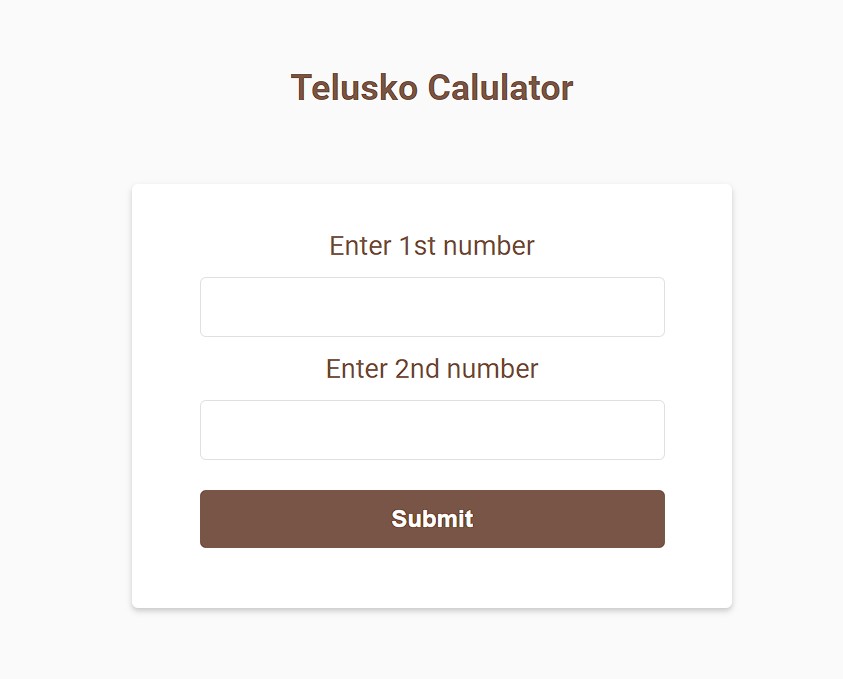
<http://localhost:8080/>

It will route to the spring boot application and try to find the controller annotated with **@Controller** and after finding the controller it will then find the method which is annotated with **@GetMapping(“URLpattern”) ,** then it will go to the method and find where flow will routed.

🡺In the method flow is routed towards the JSP page, springboot don’t understand JSP so we need to convert that JSP into servlet so we need **tomcat jasper** dependency by adding dependency in pom.xml

🡺After the JSP is converted into the servlet page page response is given back to the client machine.

Q. Write an web app to represent this webapplication and write the backend of it?



First of all we will write the controller of this web page which will route this to this web page

**@Controller**

**public class HomeController {**

**@GetMapping("/")**

**public String home() {**

**System.out.println("home called");**

**return "index.jsp";**

**}**

**}**

Above code is responsible for route the flow to the webpage , now let’s see the jsp page which will represent the above page:-

**<%@ page language="java" contentType="text/html; charset=UTF-8"**

**pageEncoding="UTF-8"%>**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<link rel='stylesheet' type='text/css' href='style.css'>**

**<meta charset="UTF-8">**

**<title>home</title>**

**</head>**

**<body>**

**<h2>Telusko Calulator</h2>**

**<form action='add'>**

**<label for='num1'>Enter 1st number </label>**

**<input type='text' id ='num1'name='num1'><br>**

**<label for='num2'>Enter 2nd number </label>**

**<input type='text' id ='num2'name='num2'><br>**

**<input type='submit' value='Submit'>**

**</form>**

**</body>**

**</html>**

As we can see in the above jsp page , in the form section we have written form **action =add,** it mean when we put the numbers and click on submit first it will pass the value of the number is query parameter and then it will try to find the add method in the controller, now again let’s see the controller:-

Link with query parameter

<http://localhost:8080/add?num1=5&num2=6>

First we will fetch the value of the number using **HttpServletRequest req**

package com.telusko.SpringJDBCEx;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import jakarta.servlet.http.HttpSession;

@Controller

public class HomeController {

@GetMapping("/")

public String home() {

return "index.jsp";

}

@GetMapping("add")

public String add(HttpServletRequest req, HttpServletResponse res,HttpSession session) {

int num1 = Integer.parseInt(req.getParameter("num1"));

int num2 = Integer.parseInt(req.getParameter("num2"));

int result =num1+num2;

session.setAttribute("result", result);

return "result.jsp";

}

}

🡺 As we can see above we have fetch the value using servlet way i.e. using **HttpServletRequest,** Now we will try to fetch the value of the number using **using** Spring way i.e. by using **Query parameter.**

<http://localhost:8080/add?num1=5&num2=6>

**🡺**As we can see the query parameter there is value of num1 and num2 we can directly use num1 and num2 in the controller

Like this

@GetMapping("add")

public String add(int num1,int num2,HttpSession session) {

int result =num1+num2;

session.setAttribute("result", result);

return "result.jsp";

}

The variable name should be same that is used in the query parameter other wise it will throw an client error. But now suppose if the variable name is not same than we can use the annotation

@RequestParam:- @RequestParam annotation enables Spring to extract input data that may be passed as a query, form data, or any arbitrary custom data.

We just need it write **@RequestParam(“Queryparameter name”)** Before the variable it will automatically allocation the query parameter value in the variable name. like this

@GetMapping("add")

public String add(@RequestParam(“num1”) int num1, @RequestParam(“num2”) int num2,HttpSession session) {

int result =num1+num2;

session.setAttribute("result", result);

return "result.jsp";

}

In the controller we have created session object , in order to send the value of result to the result jsp, let’s see the result jsp which will reflect the result of the two numbers:-

**<%@ page language="java" contentType="text/html; charset=UTF-8"**

**pageEncoding="UTF-8"%>**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<link rel='stylesheet' type='text/css' href='style.css'>**

**<meta charset="UTF-8">**

**<title>Insert title here</title>**

**</head>**

**<body>**

**<h2>Result is: <%=session.getAttribute("result")%></h2>**

**</body>**

**</html>**

IN the above we have use expression tag to fetch the value of result which is stored in the session object of the controller.

Till now we have used session to send the attribute in the JSP, we do this in the J2EE development, Now we are going to use **Model Object**

As we know in the MVC, we have Model,view,controller and Model act as a data transfer between view and controller or controller and database so in our code we are simple going to use model object.

🡺 we are simply going to add the model object in place of session and attribute like session in it as described below:-

@Controller

public class HomeController {

@GetMapping("/")

public String home() {

//System.out.println("home called");

return "index.jsp";

}

@GetMapping("add")

public String add(@RequestParam("num1")int num1,@RequestParam("num2")int num2,Model model) {

int result =num1+num2;

model.addAttribute("result",result);

return "result.jsp";

}

}

In Html , we are going to change simply curly bracket to fetch the value as shown below:-

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<link rel='stylesheet' type='text/css' href='style.css'>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

<h2>Result is:${result}</h2>

</body>

</html>

${result} is a JSTL which we have used here.

**View Resolver**

In the context of web development, particularly when working with frameworks like Spring in Java, a viewResolver is used to determine which view to render based on the controller's response. It is a key component of the MVC (Model-View-Controller) design pattern.

**Purpose:**

A viewResolver helps map a logical view name returned by the controller to an actual view (e.g., JSP, Thymeleaf, or other template engines) that can be rendered in the response.

Now suppose we have change the view from jsp to thymeleaf or other extension view technology and kept the view in other folder. And suppose we don’t return the extension of the jsp page by the controller as we have use different view technology here to solve this problem here.

We are going to use to prefix and suffix for that let’s see how we are going to use we first need to set the attribute in the application.properties.

**Prefix=view folder**

**Suffix=.jsp**

By default , eclipse search the view technology in the following folder like **src/main/webapp.** Now suppose we have make a subfolder inside the webapps then we need to define the

**spring.mvc.view.prefix=/view/**

**spring.mvc.view.suffix=.jsp**

We don’t need to put extension in the controller for the view technology view resolver will automatically search in the **application.properties** , prefix and suffix and keeps the view technology ready.

**For Static Resources (CSS, JS, Images): It's better to place them in the /static/, /public/, or /resources/ directories, and Spring will automatically serve them without needing a view resolver.**

**Or we can directly put in the webapp folder**

The controller will like this

**@GetMapping("add")**

**public String add(@RequestParam("num1")int num1,@RequestParam("num2")int num2,Model model) {**

**int result =num1+num2;**

**model.addAttribute("result",result);**

**return "result"; // it is the jsp page**

**}**

**Model and View object:-**

It seems like you're asking about the ModelAndView object in Spring MVC. The ModelAndView object is a key part of Spring MVC and allows you to handle both the model and view in a single object. Let me clarify the concept and usage of ModelAndView in Spring MVC.

**ModelAndView in Spring MVC:**

🡺In Spring MVC, ModelAndView is a class that holds both:

🡺The model data (the information you want to send to the view).

🡺The view name (the template or JSP/Thymeleaf file that will be rendered).

By using ModelAndView, you can combine both the data and the view in a single return value from your controller method. This can simplify the code and make it easier to return both the model data and the view name.

Let’s see the example here in our controller we are using model to pass the data between the controller and view but what if we pass both in the same object let’s see:-

**@GetMapping("add")**

**public ModelAndView add(@RequestParam("num1")int num1,@RequestParam("num2")int num2, ModelAndView mv) {**

**int result =num1+num2;**

**mv.addObject(“result”,result); //adding the attribute**

**mv.setViewName("result");**

**return mv;**

**}**

**Model Attribute**

@ModelAttribute binds a method parameter or method return value to a named model attribute, which is exposed to web views. Methods annotated with @ModelAttribute are invoked before the controller methods with @RequestMapping.

**Now lets’ suppose we have 10 attribute present in the JSP page and we want to transfer that attribute from JSP to database. At that time we need to create the Pojo class/entity class to store the value of the parameter and each variable is the corresponding property of the jsp page in it.**

**🡺 To transfer the value we need to create the object of entity class and try to add the value in it. But as we know we can’t create the object in the spring boot as it violates the functionality of loose coupling at that time Model attribute comes into the picture.**

Let’s see the example

View of the page

Controller

Controller with method of addalien method contains a attribute model

Client gives the request using URL adding parameter of alien class

**Convert the jsp by Tom cat jasper dependency into servlet and sent back as reponse**

Pojo class of the alien

🡺We will jsp in which we need to feed the parameter value as **ald & aname** after click on the submit , the flow of execution will go to controller

🡺 We have create the POJO class of alien which contain getter and setter of the parameters of jsp

🡺Now in Controller, we are going to use @ModelAttribute to directly store the parameter value of the jsp in the reference of the pojo class, Spring boot will automatically create the object of it.

🡺 Now we will return the view of spring boot.

🡺Let’s see the code properly.

First of all let’s see the JSP page

**<%@ page language="java" contentType="text/html; charset=UTF-8"**

**pageEncoding="UTF-8"%>**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<link rel='stylesheet' type='text/css' href='style.css'>**

**<meta charset="UTF-8">**

**<title>home</title>**

**</head>**

**<body>**

**<h2>Telusko Calulator</h2>**

**<form action='addAlien'>**

**<label for='aid'>Enter ID </label>**

**<input type='text' id ='aid'name='aid'><br>**

**<label for='aname'>Enter the name</label>**

**<input type='text' id ='aname'name='aname'><br>**

**<input type='submit' value='Submit'>**

**</form>**

**</body>**

**</html>**

Now let’s see the pojo class

**package com.telusko.SpringJDBCEx;**

**public class Alien {**

**private int aid;**

**private String aname;**

**@Override**

**public String toString() {**

**return "Alien [aid=" + aid + ", aname=" + aname + "]";**

**}**

**public String getAname() {**

**return aname;**

**}**

**public void setAname(String aname) {**

**this.aname = aname;**

**}**

**public int getAid() {**

**return aid;**

**}**

**public void setAid(int aid) {**

**this.aid = aid;**

**}**

**}**

Now let’s see the controller of the spring boot

**@Controller**

**public class HomeController {**

**@GetMapping("/")**

**public String home() {**

**//System.out.println("home called");**

**return "index";**

**}**

**@GetMapping("addAlien")**

**public String add(@ModelAttribute Alien alien) { //here @ModelAttribute is optional we can remove this annotation**

**return "result";**

**}**

**}**

The result JSP name is :-

**<%@ page language="java" contentType="text/html; charset=UTF-8"**

**pageEncoding="UTF-8"%>**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<link rel='stylesheet' type='text/css' href='style.css'>**

**<meta charset="UTF-8">**

**<title>Insert title here</title>**

**</head>**

**<body>**

**<h2> welcome to Gupta dynasty</h2>**

**<p>${alien}<p>**

**</body>**

**</html>**

**Creation of Spring MVC**

**🡺** To create the spring MVC project we need to create a maven project with catalog=internal **maven-archetyoe-webapp**

**🡺**We will create the same spring boot project using spring mvc and we will see what needs to change in the Spring MVC to work properly.

**Dispatcher Servlet:-**

DispatcherServlet acts as the Front Controller for Spring-based web applications.

So now **what is Front Controller?**

it is pretty simple. Any request is going to come into our website the front controller is going to stand in front and is going to accept all the requests and once the front controller accepts that request then this is the job of the front controller that it will make a decision that who is the right controller to handle that request.

**In case of Spring Boot we don’t need to configure the Dispatcher Servlet, whenever we write the url pattern in the browser it will automatically goes to the required controller:-**

But in case of Spring MVC we need to configure the Dispatcher Servlet in order to route the flow of execution , we need to do this configuration inside the **Web.xml**

**Before running any spring project we need to add the tomcat in the build path as a library.**

**<!DOCTYPE web-app PUBLIC**

**"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"**

**"http://java.sun.com/dtd/web-app\_2\_3.dtd" >**

**<web-app>**

**<display-name>Archetype Created Web Application</display-name>**

**<servlet>**

**<servlet-name>telusko</servlet-name>**

**<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>telusko</servlet-name>**

**<url-pattern>/</url-pattern>**

**</servlet-mapping>**

**</web-app>**

Now till now we have only provide the control of request to dispatcher servlet, now how will the dispatcher servlet will know for what request , the request will be sent to which controller for that we need to create another **XML** file in WEB\_INF with the name which is provided in the web.xml concatenated with –servlet.

For example if the name in the **web.xml** is **telusko** then corresponding xml which contains the routing information of the servlet will be **telusko-servlet.**

Now let’s see what need to be configured in **telusko-servlet** for routing.

**So we need to tell the servlet to find the class in the package using annotation, let’s see how to do it.**

To do this we need to use some tags:-

This tags will define the definition of the beans :-

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

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

**xmlns:ctx="http://www.springframework.org/schema/context"**

**xmlns:p="http://www.springframework.org/schema/p"**

**xmlns:mvc="http://www.springframework.org/schema/mvc"**

**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**

[**http://www.springframework.org/schema/mvc**](http://www.springframework.org/schema/mvc)

[**http://www.springframework.org/schema/mvc/spring-mvc.xsd**](http://www.springframework.org/schema/mvc/spring-mvc.xsd)**">**

**<ctx:component-scan base-package="com.telusko"></ctx:component-scan>**

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

**//**The above two line defines to search the package with the annotation

**</beans>**

In Spring Boot we have resolve the view by defining the prefix and suffix in the application.properties but in the SpringMVC application we need to do the same in the **XML file** of the web servlet let’s see how:-

**View Resolver**

In Spring MVC, a ViewResolver is used to map logical view names returned by the Controller to actual view objects. It provides flexibility to configure more than one view resolver. Examples of view resolvers include **InternalResourceViewResolver**, XmlViewResolver, and ResourceBundleViewResolver.

**Let’s see how to resolve the view in xml of webapp**

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

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

**xmlns:ctx="http://www.springframework.org/schema/context"**

**xmlns:p="http://www.springframework.org/schema/p"**

**xmlns:mvc="http://www.springframework.org/schema/mvc"**

**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**

[**http://www.springframework.org/schema/mvc**](http://www.springframework.org/schema/mvc)

[**http://www.springframework.org/schema/mvc/spring-mvc.xsd**](http://www.springframework.org/schema/mvc/spring-mvc.xsd)**">**

**<ctx:component-scan base-package="com.telusko"></ctx:component-scan>**

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

**//The below code is used for view resolver purpose**

**<bean class ="org.springframework.web.servlet.view.InternalResourceViewResolver">**

**<property name ="prefix" value=”/value/”></property>**

**<property name ="suffix"value=”.jsp”></property>**

**</bean>**

**</beans>**

**@RequestMapping**

This annotation is a versatile and flexible annotation that can be used with a controller (class) as well as the methods to map specific web requests with the handler methods and controllers. This annotation is part of a larger set of annotations provided by Spring Framework to define URL endpoints and simplify the development of Spring Boot applications.

**@GetMapping:-**

The @GetMapping annotation in Spring is a powerful tool for building RESTful web services. It maps HTTP GET requests to specific handler methods in Spring controllers, allowing you to define endpoints for your RESTful API and handle various HTTP requests efficiently.

**@GetMapping vs @RequestMapping**

The @GetMapping annotation is essentially a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET). This means that it simplifies the process of mapping HTTP GET requests to handler methods by eliminating the need to specify the request method explicitly.

**Spring ORM(Hibernate,Toplink,IBATIS):-**

**Advantage of using Spring ORM**

\*Less Code is required(We don’t need to provide the Transaction code or manage the code)

\*Easy to test

\*Better exception handling

\*Integrated transaction handling

In Spring ORM, we have Hibernate template which contains these methods in it to perform crud Operation like:-

🡺Save

🡺Update

🡺Insert

🡺Get

🡺loadAll

Session Factory(Interface)

HibernateTemplate

Product Dao

Datasource

(URL,username,password)

LocalSessionFactoryBean

(dataSource)

(Hibernate Properties)

(Anotated class)

**Implements**

The above picture simplify how the Spring ORM work:-

First of all we need to create the Dao class in order to connect with the database

Now we need hibernate template to perform Crud operation in the database.

Now we need Sessionfactory object to execute the hibernate template as Session factory is an interface so we need the implemented class of it i.e. **LocalSessionFactoryBean** which requires three things:-

1. dataSource
2. Hibernate Properties
3. Annotated class

First of all we have create the Maven project for Spring ORM and add these dependencies in it:-

**<dependency>**

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

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

**<version>5.2.3.RELEASE</version>**

**</dependency>**

**<dependency>**

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

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

**<version>5.2.3.RELEASE</version>**

**</dependency>**

**<dependency>**

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

**<artifactId>spring-orm</artifactId>**

**<version>5.2.3.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>mysql</groupId>**

**<artifactId>mysql-connector-java</artifactId>**

**<version>5.1.30</version>**

**</dependency>**

**<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->**

**<dependency>**

**<groupId>org.hibernate</groupId>**

**<artifactId>hibernate-core</artifactId>**

**<version>5.4.25.Final</version>**

**</dependency>**

**Let’s create the application for the SPRING ORM where we are going to create the one entity class i.e. Student and we will write the Dao class of it. Let’s see the entity class:-**

Student Entity class:-

package com.spring.orm.SpringORm.entities;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name="sudent\_details")

public class Student {

@Id

@Column(name="student\_id")

private int studentId;

@Column(name="student\_name")

private String studentName;

@Column(name="student\_city")

private String studentCity;

@Override

public String toString() {

return "Student [studentId=" + studentId + ", studentName=" + studentName + ", studentCity=" + studentCity

+ "]";

}

public int getStudentId() {

return studentId;

}

public void setStudentId(int studentId) {

this.studentId = studentId;

}

public String getStudentName() {

return studentName;

}

public void setStudentName(String studentName) {

this.studentName = studentName;

}

public String getStudentCity() {

return studentCity;

}

public void setStudentCity(String studentCity) {

this.studentCity = studentCity;

}

public Student() {

super();

// TODO Auto-generated constructor stub

}

public Student(int studentId, String studentName, String studentCity) {

this.studentId = studentId;

this.studentName = studentName;

this.studentCity = studentCity;

}

}

Now let’s see the **StudentDAO ,**Where we are going to add hibernate template which is responsible for crud operation in the hibernate:-

**package com.spring.orm.SpringORm.dao;**

**import org.springframework.orm.hibernate5.HibernateTemplate;**

**import com.spring.orm.SpringORm.entities.Student;**

**public class StudentDao {**

**private HibernateTemplate hibernateTemplate;**

**public HibernateTemplate getHibernateTemplate() {**

**return hibernateTemplate;**

**}**

**public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {**

**this.hibernateTemplate = hibernateTemplate;**

**}**

**public int insert(Student student) {**

**Integer i=(Integer)this.hibernateTemplate.save(student);**

**return i;**

**}**

**}**

Let’s see the configuration file:-

**<?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:ctx="http://www.springframework.org/schema/context"**

**xmlns:mvc="http://www.springframework.org/schema/mvc"**

**xmlns:tx="http://www.springframework.org/schema/tx"**

**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**

**http://www.springframework.org/schema/mvc**

**http://www.springframework.org/schema/mvc/spring-mvc.xsd**

**http://www.springframework.org/schema/tx**

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

**<!-- DataSource Configuration -->**

**<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">**

**<property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>**

**<property name="url" value="jdbc:mysql://localhost:3306/ineuron"/>**

**<property name="username" value="root"/>**

**<property name="password" value="Gaurav45"/>**

**</bean>**

**<!-- SessionFactory Configuration -->**

**<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">**

**<property name="dataSource" ref="dataSource"/>**

**<property name="hibernateProperties">**

**<props>**

**<prop key="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</prop>**

**<prop key="hibernate.show\_sql">true</prop>**

**<prop key="hibernate.hbm2ddl.auto">update</prop>**

**</props>**

**</property>**

**<!-- annnotated class-->**

**<property name="annotatedClasses">**

**<list>**

**<value>com.spring.orm.SpringORm.entities.Student</value>**

**</list>**

**</property>**

**</bean>**

**<!-- HibernateTemplate Configuration -->**

**<bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate">**

**<property name="sessionFactory" ref="sessionFactory"/>**

**</bean>**

**<!-- StudentDao Configuration -->**

**<bean id="studentDao" class="com.spring.orm.SpringORm.dao.StudentDao">**

**<property name="hibernateTemplate" ref="hibernateTemplate"/>**

**</bean>**

**</beans>**

**Let’s see the main application:-**

**package com.spring.orm.SpringORm;**

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

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

**import com.spring.orm.SpringORm.dao.StudentDao;**

**import com.spring.orm.SpringORm.entities.Student;**

**public class App**

**{**

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

**{**

**ApplicationContext context= new ClassPathXmlApplicationContext("config.xml");**

**StudentDao studentDao=context.getBean("studentDao",StudentDao.class);**

**Student student = new Student(1224,"gaurav Gupta","Ranchi");**

**studentDao.insert(student);**

**System.out.println("done");**

**}**

**}**

**@Transactional Annotation**

The @Transactional annotation in Spring is used to manage transaction boundaries declaratively. It ensures that a method or class is executed within a transactional context, typically for database operations.

**What it does**

* Starts a transaction before method execution.
* Commits it if the method completes successfully.
* Rolls back if an unchecked exception (RuntimeException or Error) is thrown.

Let’s create the update operation in **StudentDao,** lets see the code of StudentDao:-

package com.spring.orm.SpringORm.dao;

import javax.transaction.Transactional;

import org.springframework.orm.hibernate5.HibernateTemplate;

import com.spring.orm.SpringORm.entities.Student;

import antlr.collections.List;

public class StudentDao {

private HibernateTemplate hibernateTemplate;

public HibernateTemplate getHibernateTemplate() {

return hibernateTemplate;

}

public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {

this.hibernateTemplate = hibernateTemplate;

}

@Transactional

public int insert(Student student) {

Integer i=(Integer)this.hibernateTemplate.save(student);

return i;

}

public Student getStudent(int studentId) {

Student student=this.hibernateTemplate.load(Student.class, studentId);

return student;

}

public java.util.List<Student> getStudents(){

java.util.List<Student> students=this.hibernateTemplate.loadAll(Student.class);

return students;

}

//delete the data

@Transactional

public void deleteStudent(int studentID) {

Student student=this.hibernateTemplate.load(Student.class, studentID);

this.hibernateTemplate.delete(student);

}

//update the data

@Transactional

public void updateStudent(Student student) {

this.hibernateTemplate.update(student);

}

}

Let’s see the main app here:-

package com.spring.orm.SpringORm;

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.spring.orm.SpringORm.dao.StudentDao;

import com.spring.orm.SpringORm.entities.Student;

public class App {

public static void main(String[] args) {

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

StudentDao studentDao = context.getBean("studentDao", StudentDao.class);

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

boolean go =true;

Integer sid=null;

Student stuValue=null;

String uName=null;

String uCity=null;

while (go) {

System.out.println("Press 1 for add new Student");

System.out.println("Press 2 for diplay all Student");

System.out.println("Press 3 for get details of single student");

System.out.println("Press 4 for delete Students");

System.out.println("Press 5 for update Students");

System.out.println("Press 6 for exit");

try {

int input = Integer.parseInt(br.readLine());

switch (input) {

case 1:

//add new Student

System.out.println("enter the user Id");

sid=Integer.parseInt(br.readLine());

System.out.println("enter the User name");

uName=br.readLine();

System.out.println("enter the city");

uCity=br.readLine();

Student student = new Student(sid,uName,uCity);

studentDao.insert(student);

break;

case 2:

//display all Student

List<Student> studnetList =studentDao.getAllStudents();

for(Student a:studnetList) {

System.out.println("Student id :"+a.getStudentId());

System.out.println("Student name :"+a.getStudentName());

System.out.println("Student city :"+a.getStudentCity());

}

break;

case 3:

//get single Student data

System.out.println("Enter the Student id which you want");

sid =Integer.parseInt(br.readLine());

stuValue=studentDao.getStudent(sid);

System.out.println("Student id :"+stuValue.getStudentId());

System.out.println("Student name :"+stuValue.getStudentName());

System.out.println("Student city :"+stuValue.getStudentCity());

break;

case 4:

//delete Students

System.out.println("Enter the Student id which you want");

sid =Integer.parseInt(br.readLine());

studentDao.deleteStudent(sid);

break;

case 5:

//Update Students

System.out.println("enter the user Id");

sid=Integer.parseInt(br.readLine());

stuValue=studentDao.getStudent(sid);

sid=stuValue.getStudentId();

uName=stuValue.getStudentName();

uCity=stuValue.getStudentCity();

System.out.println("enter the name which needs to be updated");

String stuName=(br.readLine()!=null)?br.readLine():uName;

System.out.println("enter the city which needs to be updated");

String stucity=(br.readLine()!=null)?br.readLine():uCity;

stuValue.setStudentId(sid);

stuValue.setStudentCity(stucity);

stuValue.setStudentName(stuName);

break;

case 6:

go=false;

break;

default:

break;

}

} catch (Exception e) {

System.out.println("Invalid Input try with another one!!");

}

}

System.out.println("Thank you for using my application");

System.out.println("see you soon");

}

}

**What is the use of Application.properties in Spring Boot**

In **Spring Boot**, the application.properties file (or alternatively application.yml) is used to **configure application settings** in a centralized and easily manageable way. It plays a critical role in defining how your Spring Boot application behaves in different environments.

**✅ Key Uses of application.properties:**

1. **Configuring Server Properties**
2. server.port=8081
3. server.servlet.context-path=/api
   * Changes the port from default 8080 to 8081
   * Sets a base URL path for all endpoints
4. **Database Configuration**
5. spring.datasource.url=jdbc:mysql://localhost:3306/mydb
6. spring.datasource.username=root
7. spring.datasource.password=secret
8. spring.jpa.hibernate.ddl-auto=update
9. spring.jpa.show-sql=true
   * Connects your app to a database
   * Configures JPA/Hibernate settings
10. **Logging Configuration**
11. logging.level.org.springframework=INFO
12. logging.level.com.example=DEBUG
    * Controls log output for specific packages/classes
13. **Custom Application Properties**
14. myapp.feature.enabled=true
15. myapp.default.user=admin
    * You can define your own properties and inject them using @Value or @ConfigurationProperties
16. **Spring Profiles**
17. spring.profiles.active=dev
    * Activates a specific profile (e.g., dev, test, prod)
    * Allows environment-specific configuration using application-dev.properties, etc.
18. **Security Configurations**
19. spring.security.user.name=admin
20. spring.security.user.password=admin123
21. **Mail, Cache, Swagger, Scheduling, etc.**

It can also be used to configure:

* + Email server
  + Caching
  + API documentation (Swagger)
  + Task scheduling

**💡 Summary:**

application.properties is **essential for centralizing and externalizing configuration**, making your Spring Boot application easier to manage, deploy, and maintain across different environments (development, staging, production, etc.).

Let me know if you'd like examples of how to use application.yml instead, or how to load properties into beans.

**Spring Boot JPA(Java Persistence API)**

JPA stands Java persistence API, It is java specification and is used to interacte with relational database system. It is a ORM tool.

Spring boot use springBoot JPA, which is built on the top of JPA to interact with data base more efficiently.

**Difference between Hibernate and JPA**

JPA is a specification which is given by the oracle, it provides a way to right the code,it doesn’t provide any implementation where as Hibernate is the implementation of JPA which works behind the scene of the JPA.

**JPA provides us two important interfaces i.e.**

**🡺**EntityManagerFactory-FactoryClass

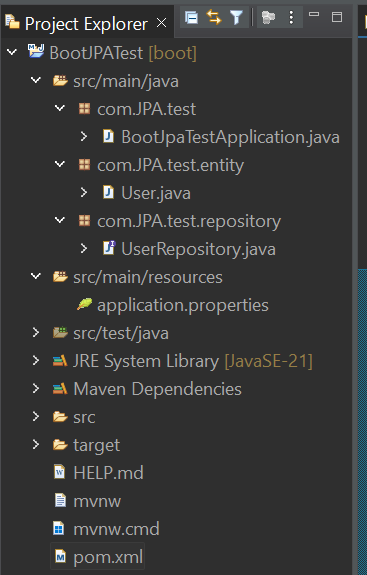
🡺EntityManager-interface

Here EntityManagerFactory will provides use EntityManager and entityManger provides us those method which is generally use for creation,updation,read and delete. Methods are **Create,Update,Read,Delete.**

**Dependency used for SpringBoot JPA- SpringBootStarter-jpa.**

Now suppose we need to do operation on **user(entity class)** class**,** we need to create the interface **UserRepository** which entends either **CRUDRepository or JpaRepository** Interface,

Now , let’s see how to integrate **Spring Data JPA,** First of all we will see the file archietecture of the project then we will see the code here:-



Above there is design for the spring data JPA:- we have added two dependency in our project:-

**Spring boot starter JPA**

**MySql Plugin**

**For database connectivity we added properties in the application.properties:-**

**spring.datasource.name=ineuron**

**spring.datasource.url=jdbc:mysql://localhost:3306/ineuron**

**spring.datasource.username=root**

**spring.datasource.password=Gaurav45**

**spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver**

**spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MySQL8Dialect**

**spring.jpa.hibernate.ddl-auto=update**

We have created the Entity class:-

package com.JPA.test.entity;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

@Entity

public class User {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private Integer id;

private String name;

private String city;

private String status;

public User() {

super();

// TODO Auto-generated constructor stub

}

public User(Integer id, String name, String city, String status) {

super();

this.id = id;

this.name = name;

this.city = city;

this.status = status;

}

@Override

public String toString() {

return "User [id=" + id + ", name=" + name + ", city=" + city + ", status=" + status + "]";

}

public User(String city) {

super();

this.city = city;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public String getStatus() {

return status;

}

public void setStatus(String status) {

this.status = status;

}

}

Now we have created an Interface which will extends CRUDRepository or JPARepository Let’s see the code:-

public interface UserRepository extends CrudRepository<User, Integer>{

}

Now let’s see how we are going to inject value inside the JPA using main class:-

**CRUD operation in Spring DATAJPA**

For create operation in UserRepository we have two methods:-

**Save()🡪** In this we will pass the one instance of the entity class

**saveAll()-** In this we will pass the list of instance of entity class or if we have more than one instance of entity class at that time we are going to use SaveALL() method.

Above we have seen how to use the save method , but here we are going to see how to use saveAll() method in the Spring data JPA.

**For Save operation:-**

@SpringBootApplication

public class BootJpaTestApplication {

public static void main(String[] args) {

ApplicationContext context =SpringApplication.run(BootJpaTestApplication.class, args);

UserRepository userRepository=context.getBean(UserRepository.class);

User user = new User();

user.setCity("Ranchi");

user.setName("Gaurav");

user.setStatus("I an java devloper");

User user1 = userRepository.save(user);

System.out.println(user1);

}

}

**For SaveAll operation we have code like:-**

**@SpringBootApplication**

**public class BootJpaTestApplication {**

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

**ApplicationContext context =SpringApplication.run(BootJpaTestApplication.class, args);**

**UserRepository userRepository=context.getBean(UserRepository.class);**

**User user = new User();**

**user.setCity("Ranchi");**

**user.setName("Gaurav");**

**user.setStatus("I an java devloper");**

**User user1 = new User();**

**user1.setCity("Ramgarh");**

**user1.setName("Gautam");**

**user1.setStatus("Python Developer");**

**List<User> users= List.of(user,user1);**

**Iterable<User> result=userRepository.saveAll(users);**

**result.forEach(System.out::print);**

**System.out.println(user1);**

**}**

**}**

Now suppose we want to update the name of one user from Gautam to Gautam gupta, let’s see how we are going to do it in the Spring boot data jpa.

Now first of all we need to find that user by id and then we need to update it let’s see how to find the user we have two method i.e. **findById** and **findAll**

**Let’s see the code here:-**

**@SpringBootApplication**

**public class BootJpaTestApplication {**

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

**ApplicationContext context =SpringApplication.run(BootJpaTestApplication.class, args);**

**UserRepository userRepository=context.getBean(UserRepository.class);**

**Optional<User> optional=userRepository.findById(5);**

**User user=optional.get();**

**user.setName("Gautam gupta");**

**User result=userRepository.save(user);**

**System.out.println(result);**

**}**

**}**

Here first of all we have got the data using **findById,**it will provide the optional data , from that optional data we have got the actual data of the entity class

In that class we simple update the name and pass that object in the save method of **UserRepository** to process it , it automatically get updated in the database with the new name in the database.

**Delete operation in Spring Data JPA**

We have different method which is generally used to delete the data from the data base using Spring Data JPA

userRepository.deleteById(1L); // Deletes by primary key

userRepository.delete(user); // Deletes by entity object

userRepository.deleteAll(); // Deletes all records

userRepository.deleteAll(users); // Deletes a list of entities

let’s see the code here:-

@SpringBootApplication

public class BootJpaTestApplication {

public static void main(String[] args) {

ApplicationContext context =SpringApplication.run(BootJpaTestApplication.class, args);

UserRepository userRepository=context.getBean(UserRepository.class);

Optional<User> optional=userRepository.findById(4);

User user=optional.get();

userRepository.delete(user);

userRepository.deleteById(7);

userRepository.deleteAll();

}

}

**Custom FinderMethod**

A **finder method** is a query method defined in a Spring Data JPA repository interface that retrieves data based on method naming conventions or custom queries. You don’t need to write the actual JPQL or SQL; Spring Data JPA generates the query automatically by parsing the method name or you can provide custom queries.

**How to create custom finder methods?**

#### 1. **Using Method Naming Convention**

You can define methods in your repository interface following a naming pattern based on the entity fields:

We don’t have to provide the implementation of the custom finderMethod, SpringDataJPA will automatically create the method for us:-

**Let’s see how to code in the crudRepository interface:-**

**public interface UserRepository extends CrudRepository<User, Integer>{**

**List<User> findByName(String name);**

**List<User> findByCity(String city);**

**}**

Now let’s see the main Application :-

@SpringBootApplication

public class BootJpaTestApplication {

public static void main(String[] args) {

ApplicationContext context =SpringApplication.run(BootJpaTestApplication.class, args);

UserRepository userRepository=context.getBean(UserRepository.class);

List<User> optional=userRepository.findByCity("Ranchi");

optional.forEach(System.out::print);

}

}

For Custom finder method we don’t need to write the implementation of the finder we just simply need to write the method name in most meaning way so that the spring data will understand it properly and provide the implementation of it.

**Exception Handling in Spring Boot**

To handle global exceptions in Spring Boot, you can use @ControllerAdvice or @RestControllerAdvice to create a centralized exception handling mechanism. Here's a basic setup:

1. Define a class with @ControllerAdvice or @RestControllerAdvice.
2. Annotate methods with @ExceptionHandler to catch specific exceptions.
3. Return a suitable ResponseEntity or custom response object.

You can view the code example above for a simple implementation. Let me know if you need JSON response formatting, logging integration, or more advanced customization like error codes or validation handling.

**import org.springframework.http.HttpStatus;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.ControllerAdvice;**

**import org.springframework.web.bind.annotation.ExceptionHandler;**

**import org.springframework.web.bind.annotation.RestControllerAdvice;**

**// Use @RestControllerAdvice if you want to return JSON responses directly**

**//@RestControllerAdvice**

**@ControllerAdvice**

**public class GlobalExceptionHandler {**

**// Handle specific exception**

**@ExceptionHandler(ResourceNotFoundException.class)**

**public ResponseEntity<String> handleResourceNotFound(ResourceNotFoundException ex) {**

**return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);**

**}**

**// Handle generic exception**

**@ExceptionHandler(Exception.class)**

**public ResponseEntity<String> handleGenericException(Exception ex) {**

**return new ResponseEntity<>("An unexpected error occurred: " + ex.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);**

**}**

**// You can add more handlers here for different types of exceptions**

**}**

**// Example of a custom exception class**

**class ResourceNotFoundException extends RuntimeException {**

**public ResourceNotFoundException(String message) {**

**super(message);**

**}**

**}**