SPRING FRAMEWORK

Simplifying Enterprise Java Development

ABSTRACT

"Spring and Spring Boot: Mastering Modern Java Development" is a concise, practical guide designed to help developers efficiently build robust and scalable Java applications. This book covers the essentials of the Spring framework, including Inversion of Control (IoC), dependency injection, and Spring MVC, alongside Spring Boot's features like auto-configuration, embedded servers, and starter templates.”

Dinesh Sripathi
Java Full Stack Trainer & Developer

Spring

Spring :

It is used for the dependency injection. In spring we will have a IOC Container which will store the objects and make them loosely coupled instead of hard coupling.

Hard Coupling :

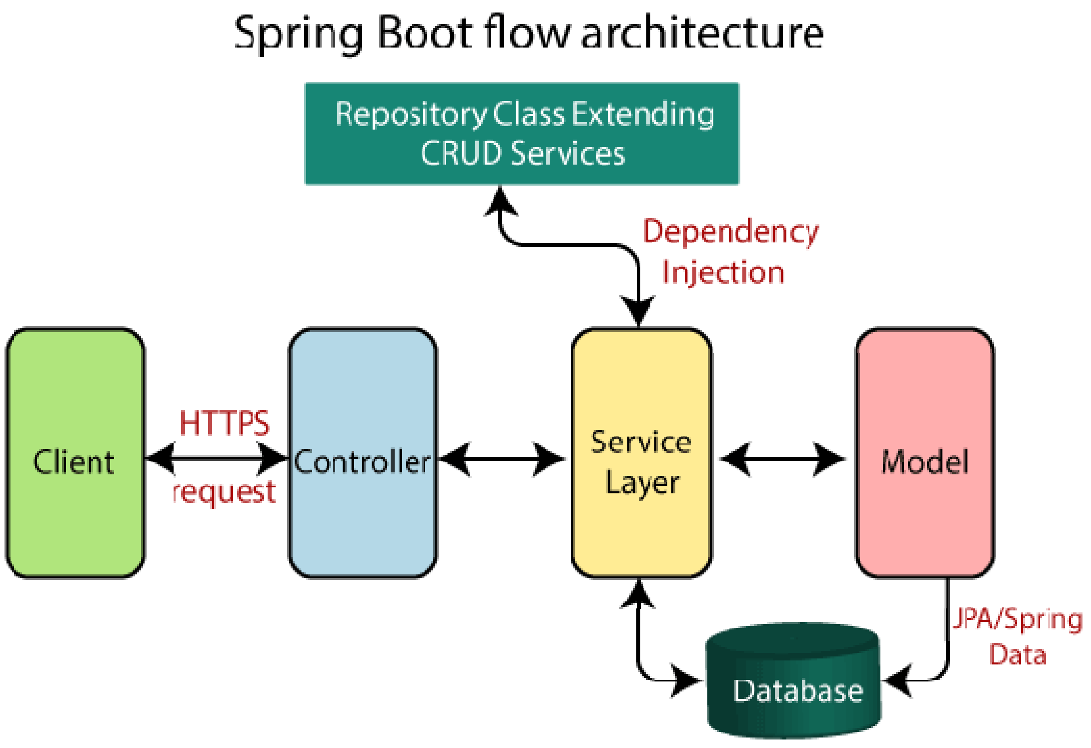
If two or more components are dependent on each other then it is called as hard coupling.

Example :

Think that you are having a interface A and class B, now you implemented the A to B. While object creation we can use the parent reference object to assign the class B space. Here we use the new keyword which we are creating the hard object creation. Here the B is completely based on A.

IOC container :

Here we define which classes will have the objects then it will create the object to them and provides to us. It will look after the object like creating deleting…



Getting started with the spring :

1. *Create a simple maven project.*
2. *Go to maven repo and search for spring core*
3. *Copy and paste the dependency In pom.xml file.*

Syntax :

```
<dependencies>
        <dependency>
                Spring-core dependency
        </dependency>
</dependencies>
```

Example :

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
    <dependency>
```

```
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>6.1.6</version>
        </dependency>
</dependencies>
```

4. *Also get the spring context from maven repo and place them in the dependencies.*

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-context
-->
<dependency>
   <groupId>org.springframework</groupId>
     <artifactId>spring-context</artifactId>
     <version>6.1.6</version>
</dependency>
```

NOTE: It is always a best practice to take the same versions when using them.

Application Context in spring :

Application Context is the pillar for the spring as it takes care of the configuration and the dependency injection (Object creation A.K.A Beans).

1. To implement the application context we use the interface ApplicationContext.

   Syntax :

   ApplicationContext AppContextObject = new classPathXmlApplicationContext(" TheBeansFile.xml");

   Example :

   ```
   ApplicationContext appContext = news
   ClassPathXmlApplicationContext("Beans.xml");
   ```

   ApplicationContext : It is an interface and it was implemented by the classPathXmlApplicationContext class.
2. Here we can mention that we can work with the xml or the annotations.
3. ***Configuration using XML :***
   a. *Go to resources*
   b. *Create a xml file.*
   c. *Go to google and search for spring xsd.*

Example :

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

</beans>
```

   d. *Here we need to create a bean tag inside the beans.*

   Syntax :

   <bean id = "unique for Bean To Bean" class = "complete class path"></bean>

Example :
```
<bean id ="iphone" class = "com.CoreSpring.FirstBeans.Iphone"></bean>
   <bean id ="redmi" class = "com.CoreSpring.FirstBeans.Redmi"></bean>
```

NOTE : Writing the bean tag in xml will create a object for the class automatically.

e. *To use the bean(Object) we have to send the id while calling for the bean. To get the bean we use the getBean() which will return the object class, If you want to work with the bean as object then we have to type convert the object.*

   Syntax :

   Object object = appContextObject.getBean("id");

Example :
```
Iphone iphone = (Iphone)appContext.getBean("iphone");
iphone.receiveMessage("Hello World");
```

f. *If you don't want to do the type casting then we have to send the another argument along with the id of bean.*

   Syntax :

   Object object = appContextObject.getBean("id", className.class);

Example :
```
Redmi redmi = appContext.getBean("redmi", Redmi.class);
redmi.receiveMessage("Hello Redmi");
```

NOTE : In Spring by default it follows the singleton design pattern. Means it will create a bean for once and it will be used multiple times.

Tight Coupling Example :
```
WhatsApp app1 = new Iphone();
WhatsApp app2 = new Redmi();

app1.sendMessage("Hello Iphone");
app2.sendMessage("Hello Redmi");
```

In the above example as you can see we are hard coding the object creation with the help of new and developer has to do it by himself. The app1 and app2 are tightly coupled to the object.

Loose Coupling Example :
```
WhatsApp app = appContext.getBean("iphone", WhatsApp.class);
    app.receiveMessage("Hello Iphone");

    app = appContext.getBean("redmi", WhatsApp.class);
    app.receiveMessage("Hello Redmi");
```

Data Injection :

Here we inject the values from the xml instead of the using the constructor. As the spring is taking care of the Objects creation we can't use the constructor hence we have to pass the values to the spring to assign them.

Steps to follow for data injection for primitive Types :

1. To pass the values to the variable we have to write a set of properties inside the bean tag.

Syntax :

```
<bean id = "uniqueValue" class = "classPath">
        <property name = "varName1" value =
"value/Data"></property>
        <property name = "varName2" value =
"value/Data"></property>
        .
        .
        .
        <property name = "varNamen" value =
"value/Data"></property>
</bean>
```

Example :

```
<bean id = "student" class="com.CoreSpring.FirstBeans.Student">
        <property name="id" value ="001"/>
        <property name="name" value ="Raj"/>
        <property name="marks" value="90.5"/>
</bean>
```

NOTE : The dependency injection will be invoked during the bean(object) creation. It will always looks for the setter method. Hence we have to use the setter method if not it will throw an error.

2. To pass the values to the variable we use another way which is from constructors. Here we use the constructor-arg and apart from it everything is same.

Syntax :

```
<constructor-arg name = "uniqueValue" value =
"value/Data"></constructor-arg>
        .
        .
        .
<constructor-arg name = "unqiueValue" value = "data">
</constructor-arg>
```

Example :

```
<bean id = "student" class="com.CoreSpring.FirstBeans.Student">
        <constructor-arg name="id" value ="001"/>
        <constructor-arg name="name" value ="Raj"/>
        <constructor-arg name="marks" value="90.5"/>
</bean>
```

NOTE: If you are using the constructor-arg we have to use the constructors to initialize the variables.

Steps to follow for data injection for Reference/Object Types :

1. If you want to initialize the reference object we have to create a bean for the reference class then we have to initialize it.
2. In property tag of the tag we use the ref property.

Syntax :

```
<constructor-arg/ property name = "uniqueId"  ref = "beanName" />
```

Example :

```
<bean id="student" class="com.CoreSpring.FirstBeans.Student">
        <constructor-arg name="id" value="001"/>
        <constructor-arg name="name" value="Raj"/>
        <constructor-arg name="marks" value="90.5"/>
        <constructor-arg name="address" ref="address" />
</bean>
```

```
<bean id="address" class="com.CoreSpring.FirstBeans.Address">
        <property name="street" value="No street"/>
        <property name="city" value="This city"/>
</bean>
```

NOTE : It will make hard if we forget to create a bean for the reference class then it will throw an error. Hence we use the auto wiring concept as it will be difficult to create bean for each and every class.

Auto-wiring :

It is going to bind the dependency objects automatically. Here we are going to mention the beans and the spring will take care of the reference by itself.

Syntax :

```
<bean id = "uniqueId" class = "qualifiedName" autowire = "type">
</bean>
```

Here auto-wiring can be done in 3 ways.

1. ***byType*** :
   a. Spring will look for a bean of the same type required by the dependent bean and inject it. If there are multiple beans of the same type, Spring may throw an error unless you specify additional qualifiers.

      Syntax :

      ```
      <bean id = "uniqueId" class = "qualifiedName" autowire =
      "byType">
              <property name = "varName" value = "data" />
      </bean>
      ```

      Example :

      ```
      <bean id="student" class="com.CoreSpring.FirstBeans.Student"
      autowire="byType">
              <property name="id" value="001"/>
              <property name="name" value="Raj"/>
              <property name="marks" value="90.5"/>
      </bean>

      <bean id="address" class="com.CoreSpring.FirstBeans.Address">
              <property name="street" value="No street"/>
              <property name="city" value="This city"/>
      </bean>
      ```

NOTE : To work with the byType autowire we have to use the property and also the name of the reference class has to be same type as mentioned in the bean or else it will throw exception.

2. ***Byname***
   a. Spring will look for a bean with the same name as the property or constructor argument and inject it.

      Syntax :

      ```
      <bean id = "uniqueId" class = "qualifiedName" autowire =
      "byname">
              <property name= "varName" value = "data" />
      </bean>
      ```

      Example :

      ```
      <bean id="student" class="com.CoreSpring.FirstBeans.Student"
      autowire="byName">
              <property name="id" value="001"/>
      ```

```
        <property name="name" value="Raj"/>
        <property name="marks" value="90.5"/>
    </bean>

    <bean id="address" class="com.CoreSpring.FirstBeans.Address">
        <property name="street" value="No street"/>
        <property name="city" value="This city"/>
    </bean>
```

NOTE : The name of the reference bean has to be same as mentioned in the main bean.

3. **constructor**
    a. Spring will look for a constructor and automatically pass dependencies as arguments to that constructor.
    Syntax :

```
        <bean id = "uniqueId" class = "qualifiedName" autowire =
"constructor">
            <constructor-arg name = "varName" value = "data" />
        </bean>
```

Example :

```
    <bean id="student" class="com.CoreSpring.FirstBeans.Student"
    autowire="constructor">
        <constructor-arg name="id" value="001"/>
        <constructor-arg name="name" value="Raj"/>
        <constructor-arg name="marks" value="90.5"/>
    </bean>

    <bean id="address" class="com.CoreSpring.FirstBeans.Address">
        <property name="street" value="No street"/>
        <property name="city" value="This city"/>
    </bean>
```

NOTE : To work with the constructor we have to use the constructor-arg or else it will not work.
NOTE : It must contain the getters and setters method.

Steps to follow for data injection using annotations :

1. To use the autowire in java we can use the annotation @autowire.
2. Generally we have to activate the support for the autowire.
3. To activate it we have to go to xsd file. We have to use the annotation config file with it.
   Example :

```
<?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
definitions here -->

</beans>
```

4. We have to enable the annotation support. By default the annotation support is disabled. To enable use the tag <context:annotation-config></context : annotation-config>

   Syntax :

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

   Example :

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

5. We have to use the autowire annotation at the setter side.

   Syntax :

   ```
   @Autowired
   Public void setMethod(){
           //block of code
   }
   ```

   Example :

   ```
   @Autowired
   public void setAddress(Address address) {
           this.address = address;
   }
   ```

6. When you are having more than one bean then we will get an exception as the autowired can use only one bean to auto-configure. To remove this ambiguity we use the annotation qualifier.

   Syntax :

   ```
   @Autowired
   @Qualifier("beanId")
   Public void setterMethod(){
           //block of code
   }
   ```

NOTE : Autowired will look for byType by default, if there is any ambiguity then it will look for byName.

NOTE : If your class is containing the variables and setters and getters then it is called as POJO (plain old java object)

Data Injection for the collection :

To insert the data into the collection we have to use the property tag and inside it we have to use the collection tag and then go for the value tag.

Syntax :

```
<property name = "varName">
        <list>
                <value>data</value>
                <value>data</value>
                .
                .
        </list>
</property>
```

Example :

```
<property name = "hobbies">
        <list>
                <value>Cricket</value>
                <value>Chess</value>
                <value>dance</value>
        </list>
</property>
```

Note : For every collection we will have the tag in xml. There is a slight minor thing which we need to use for the map.

Syntax :

```
<property name = "value">
        <map>
                <entry key = "keyName" value ="data"></entry>
                <entry key = "keyName" value ="data"></entry>
                <entry key = "keyName" value ="data"></entry>
        </map>
</property>
```

Example :

```
<property name = "hobbies">
        <map>
                <entry key = "key1" value = "4"></entry>
                <entry key = "key2" value = "42"></entry>
                <entry key = "key3" value = "44"></entry>
        </map>
</property>
```

NOTE : To insert the reference we use the value-ref property.

TIP : We have to externalize the xml code to make it more independent and developer friendly. To do that we have to create a file called application and need to save it with the extension properties.

1. To do that create a application.properties file.
2. We use the Key, values to store the data into application.properties.
   a. **Syntax :**

      Key = value

      Example :

```
student.id = 1
student.name = "Raj Raj"
```

3. Use the property placeHolders to initialize the values.
   a. **Syntax** :

      ${key}
4. But the thing is to activate it we have to use the context:property-placeholder to externalize the data.
   a. **Syntax** :

      <context:property-placeholder location = "classpath : filename" />

      Example :

```
<context:property-placeholder location = "classpath:application.properties"/>
```

NOTE: We have to save the file inside the resources folder.

5. To use the place holder we are using the ${value}.
   a. **Syntax** :

      <property name = "variableName" value = "${key}"></property>

      Example :

```
<property name="name" value="${student.name}"/>
```

No XML Configuration :

From now on we will use the annotations to work with spring.

1. To work with the annotation we have use the context:property-placeholder

2. To inject values into the variables we use the @Value annotation.
    a. **Syntax** :

        @Value("data")
        Private dataType varName;

    Example :

```
@Value("34")
private int id;
@Value("Teks")
private String name;
```

3. We are going to use the @Component to create a bean automatically. Instead of the
    a. **Syntax** :

        @Component
        Public class ClassName{
            //class contents
        }

    Example :

```
@Component
public class Student {
```

4. To mention that we are having beans in the project we need to use the component-scan tag.
    a. **Syntax** :

        <context:component-scan base-package = "packageName"></context:component-scan>

    Example :

```
<context:component-scan base-package="com.CoreSpring.FirstBeans"></context:component-scan>
```

NOTE : When you are using the @Component annotation it will create a default bean with the same name as className but in lower characters. If you want to specify the id then we have to mention the name after the annotation.

Syntax :

    @Component("idName")
    Public class ClassName{
        //body of the class
    }

Example :

```
@Component("Student")
public class Student {
 //body of the class
}
```

A brief steps for the no xml configuration :

1. In xml we need to use the below tags
    a. Use the <context:annotation-config></context:annotation-config> to enable the annotation.
    b. Use the <context:property-placeholder></context:property-placeholder> to enable the externalize the values
    c. Use the <context:component-scan base-package="parentPackage"></context:component-scan>

Creating the spring application without xml file :

1. We need to change the ClassPathXmlApplicationConfiguration to the annotation base.
    a. **Syntax** :

    ```
    ApplicationContext contextObject = new
    AnnotationConfigApplicationContext( configurationClass.class
    );
    ```

    Example :

    ```
    ApplicationContext appContext = new
    AnnotationConfigApplicationContext(ConfigurationFile.class);
    ```

2. As the xml file is actually acting as the configuration file and we are removing it hence it will throw an exception.
3. To remove the exception and to carry with the configuration we have to create a new class file and we have to use the annotation to it.
    a. **Syntax** :

    ```
    @Configuration
    Public class ClassName{
            //body of the class
    }
    ```

4. Now we have to apply the configurations here.
    a. To scan the components we use the @ComponentsScan annotation.
        i. **Syntax** :

        ```
        @Configuration
        @ComponentScan( basePackages = "basePackage" )
        Public class className{
                //body of the class
        }
        ```

        Example :

        ```
        @Configuration
        @ComponentScan( basePackages = "com.CoreSpring")
        public class ConfigurationFile {

        }
        ```

    b. To use the externalize values we use the PropertySource("classpath : .propertiesName").
        i. **Syntax** :

        ```
        @Configuration
        @ComponentScan( basePackages = "basePackage")
        @PropertySource( "classpath: propertyName" )
        Public class className{
                //body of the class
        }
        ```

        Example :

        ```
                @Configuration
            @ComponentScan( basePackages = "com.CoreSpring")
            @PropertySource("classpath:application.properties")
        public class ConfigurationFile {

            }
        ```

5. Using the autowired annotation to link the address.
    a. **Syntax** :

```
@Autowired
@Qualifier( “Name”) //optional
Private type ReferenceObject;
```

Example :

```
@Value("${student.id}")
private int id;
@Value("${student.name}")
private String name;

@Autowired
@Qualifier("address")
private Address address;
```

6. If you want to have a custom object we are going to use the @Bean. Here we are creating the object but the spring will take care of the dependency injection.
   a. **Syntax** :

```
@Bean
Public ClassName methodName{
        Return new constructor( );
}
```

Example :

```
@Bean
public Student getStudent() {
        return new Student();
}
```

NOTE : When you are using the @Bean there is no requirement for using the @Component annotation. To create a bean we have to use the methodName as bean id.

Example :

```
Student student = appContext.getBean("getStudent", Student.class);
```

NOTE : You create your own bean id using the name property.

Syntax :

```
@Bean( name = “value”)
Method( ){
        //body
}
```

Example for configuration :

```
@Bean(name = "student")
public Student getStudent() {
        return new Student();
}
```

Example for mainMethod :

```
Student student = appContext.getBean("student",
Student.class);
```

NOTE : We can give multiple names to the bean with a curly brace.

Syntax :

```
@Bean(name = { “name1”, “name2”} )
Method(){
        //body
}
```

Example :

```
@Bean(name = {"student", "student1"})
public Student getStudent() {
        return new Student();
}
```

NOTE : We are going to two ways to create beans

1. ***BeanFactory*** : lazy loading
2. ***ApplicationContext*** : Eager Loading

JDBC using Spring :

Normal JDBC steps :

It follows the same steps as JDBC.

1. *Loading driver*
2. *Registering driver*
3. *Connecting to database*
4. *Creating statement*
5. *Executing query*
6. *Result set will generate*

Steps to work with the JDBC :

1. Create a maven project
2. We need to get the below dependencies.
   a. *Get the spring core and spring context*
   b. *Spring JDBC*
   c. *MySQL Connector*
3. Here we require two classes to work with the JDBC.
   a. ***Driver Manager Data Source :***
      i. Contains the database information.
      ii. It will have four values which we needs to provide
         1. *URL*
         2. *UserName*
         3. *Password*
         4. *DriverType*

Example :

```
<bean id = "dmds" class =
"org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value =
"com.mysql.cj.jdbc.Driver"></property>
<property name="url" value = "jdbc:mysql://localhost:3306/tcs"></property>
<property name ="userName" value="root"></property>
<property name="password" value="0000"></property>
</bean>
```

   b. ***Spring JDBC template :***
      i. Uses the DriverManagerDataSource

Example :

```
<bean id ="jdbcTemplate" class =
"org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="dmds"></property>
</bean>
```

4. Now we need to connect the database but the spring will take of it. We have

to getBean of the JdbcTemplate.

Syntax :

```
ApplicationContext appContextObject = new
ClassPathXmlApplicationContext( "xmlFileName");
JdbcTemplate template = appContextObject.getBean("Id/Name",
className.class);
```

Example :

```
ApplicationContext appContext = new
ClassPathXmlApplicationContext("beanJDBC.xml");
JdbcTemplate template = appContext.getBean("jdbcTemplate",
JdbcTemplate.class);
```

5. To insert data into the database we use the update method.
    a. **Syntax** :

        ```
        templateObject.update("SQL");
        ```

        Example :

        ```
        int rows = template.update("Delete from student where id = 6");
        ```

CRUD Operations :

1. For create, update, delete we use the update method.
2. ***For Create :***
    a. We use the insert command to create.
    b. **Example** :

        ```
        int rows = template.update("insert into student values(6,
        'Stream')");
        ```
3. ***For Update :***
    a. We use the update command to update the table.
    b. **Example** :

        ```
        int rows = template.update("update student set name = 'devil'
        where id = 3");
        ```
4. ***For Delete :***
    a. We use the delete command to delete the content from the table.
    b. **Example** :

        ```
        int rows = template.update("Delete from student where id = 6");
        ```
5. ***For Read :***
    a. For read we need to catch either a single record or a multiple set of records. To handle it we are having two methods.
        i. jdbcTemplate.queryforobject ("SQL Query", rowmapper); - returns single object
        ii. jdbcTempalte.query("SQL_Query", rowMapper) – return a list of objects

NOTE : Why do we need the row mapper to map the data in the database.

b. *Creating Row Mapper :*
    i. To create row mapper we have to create a class.
    ii. Implement the RowMapper interface to the class.

        Syntax :

        ```
        Classname implements
        RowMapper<mapperClass>{
                //body of the class
        }
        ```
    iii. It will have an implementation method which will contain the result set parameter and rowNumber.
    iv. We have to use the result set methods to retrieve the data.

NOTE : The template.update will return the no.of rows that were effected

i. Jdbc Template QueryForObject :

Example :

```
RowMapper rowMapper = new RowMapper() {
                        @Override
                        public Student mapRow(ResultSet resultSet, int rows) {
                                Student student = new Student();
                                System.out.println("inside the mapRow method");
                                try {
                                        int id = resultSet.getInt("id");
                                        String name = resultSet.getString("name");
                                        System.out.println(id +" " + name);
                                        student.setId(id);
                                        student.setName(name);
                                        System.out.println(student);
                                } catch (SQLException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                                return student;
                        }
                };

           Student student = template.queryForObject("select * from student where
id = 7", rowMapper);
           System.out.println(student);
```

ii. jdbc Tempate Query :

Example :

```
                    RowMapper rowMapper = new RowMapper() {
                        @Override
                        public Student mapRow(ResultSet resultSet, int rows) {
                                Student student = new Student();
                                System.out.println("inside the mapRow method");
                                try {
                                        int id = resultSet.getInt("id");
                                        String name = resultSet.getString("name");
                                        System.out.println(id +" " + name);
                                        student.setId(id);
                                        student.setName(name);
                                        System.out.println(student);
                                } catch (SQLException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                                return student;
                        }
                };

                ArrayList<Student> students = new ArrayList<Student>();
           students =  (ArrayList<Student>) template.query("select * from student",
rowMapper);
           System.out.println(students);
```

Spring Boot

Spring Boot :

1. In spring we are having the applicationContext which is an interface used to getBean. In springBoot we use the ConfigurableApplicationContext to get the beans.
    a. **Syntax** :

       ConfigurableApplicationContext context = SpringApplication.run(className.class, args);

       Example :

```
ConfigurableApplicationContext context =
SpringApplication.run(DemoMvcApplication.class, args);
```

2. When using the getBean method we don't need to use the id to identify the bean we just have to say className.class to it, it will take care of it.
    a. **Syntax** :

       Context.getBean(className.class);

       Example :

```
Student student = context.getBean(Student.class);
```

NOTE : We have to use the component annotation to create the bean for the class.

Syntax :

@Component
Public class beanClassName

Example :

```
            @Component
        public class Student {

            }
```

NOTE : When you run the application we have to run it as spring boot App. Everytime restart the server.

API's :

The API will take the request from the client and will send the request to the DATABASE. The response provided by the database will be carried to the client. Here the api will work as a bridge between the database and client.

In backend we can write the api's in two ways.

1. **REST(Representation of State Transfer) :** *The latest form of writing the API's.*
2. **SOAP :** *It is an xml configuration and it is a legacy part*

In API's we are having 4 types of data calls.

1. **GET** :
   To get the data from the database
2. **POST** :
   Used to When you are creating new row/ record in the database
3. **PUT/ PATCH :**
   Used to update the records.
4. **DELETE :**
   We are going to delete the records in database.

We are going to implement the API's in Model View Controller (MVC).

Model View Controller :

Model – The request that was sent by the user
View – The Response that was sent to the user

Controller – Responsible for handling the request and response.

NOTE : The response will be in form of jsp. Hence we have to save it inside the main > webapp folder.

To create a simple MVC follow the below steps :

1. Goto new project -> select spring starter project
   a. When selecting the project dependencies we have to select the web application.
2. Run as the SpringApp.
3. We need to create a webapp inside the src > main folder
4. Create a jsp file inside the webapp and here we have to use the localhost address.

Creating controller to accept the request :

1. Create a Plain Old Java Object (POJO).
2. Use the annotation @Controller to the controller_Pojo class.
   a. **Syntax** :

```
@Controller
Public class className{
        //body
}
```

Example :

```
                @Controller
        public class homeController {
                        //body of the class
                }
```

3. Here the dispatch servlet will send the request and we have to use the @RequestMapping annotation to work with the request handling.
   a. Syntax :

```
@RequestMapping("name")
Public returnType methodName(){
        //body of the method
}
```

Example :

```
        @RequestMapping("home")
        public String home() {
    return "home.jsp";
   }
```

4. We need to return the response when the request raised Here we return the jsp file.
   a. Syntax :

```
@RequestMapping("name")
Public String methodName( ){
        Return "jspName.jsp";
}
```

Note : It will download the jsp file. As we are not providing how to handle the response.

5. We have to provide the response body annotation.
   a. **Syntax** :

```
@RequestMapping("home")
@ResponseBody
```

```
Public String method(){
    Return “jspFileName.jsp”;
}
```

Example :

```
    @RequestMapping("home")
    @ResponseBody
    public String home() {
   return "home.jsp";
    }
```

NOTE : It will return the response body by instead of the response text we want to return the jsp page.

6. Here we need to work with the server and we have to mention to the server we are actually mentioning the jsp file which is located in the webapp to do that we have to use an extension tomcat jasper dependencies.

Configuring the spring web app using the application properties :

1. Here we are going to the use the application.properties file to configure the spring application.
2. We can simply return the jsp file at the controller side without using the .jsp. We can even change the location of the file without any issues inside the webapp
3. To do that we have to configure the view .
    a. **Syntax** :

```
spring.mvc.view.prefix = /location/
spring.mvc.view.suffix = .jsp
```

Examle :

```
        spring.mvc.view.prefix = /WEB-INF/
      spring.mvc.view.suffix = .jsp
```

Controller Example :

```
@Controller
public class homeController {

    @RequestMapping("/home")
    public String home() {
        System.out.println("Request mapped and sending the jsp file...");
   return "home";
  }
}
```

Accepting the data from the client :

1. When the client is passing the request along with the data then the request dispatcher will take care of it.
    a. **Syntax** :

```
Public String home( HttpServletRequest reqObject){
        //body
}
```

Example :

```
    @RequestMapping("/home")
    public String home(HttpServletRequest request) {
        String name = request.getParameter("name");
        System.out.println("Request mapped and sending the jsp
file..."+name);
```

```
        return "home";
    }
```

2. At the url side the client sent data will look like as mentioned below.
   Syntax :
   Localhost:8080/pageName?variable=value
   Example :
   http://localhost:8080/home?name=Wizards
3. One servlet of the one jsp can be shared with the another servlet jsp.
4. To share the data between servlets we can use either the setAttribute method using the session.
   a. **Syntax** :
      HttpSession sessionObject = reqObject.getSession( );
      Session. setAttribute(key, value);
   Example :

```
                String name = request.getParameter("name");
        HttpSession session = request.getSession();
            session.setAttribute("user", name);
```

5. We have to use the placeholder to use the java values that was set in the session.
   a. **Example** :

```
<body>
    hello Wizards, Welcome ${user}
</body>
```

Implementing the MVC in spring boot :

1. Generally when you are working with the spring framework we need not have to mention everything like the configuration. Everything will be taken care by the spring it self.
2. Hence here we are going to get the client data directly when client sending the data.
3. Also spring will provide us the session object as well.
   a. **Syntax** :

```
@Controller
Public class className{
        @RequestMapping("name")
        Public String methodName( String value, HttpSession
sessionObject){
                SessionObject.setAttribute( key, value );
                Return "jspFileName";
        }
}
```

   Example :

```
        @RequestMapping("/home")
        public String home(String name, HttpSession session) {
                session.setAttribute("user", name);
                System.out.println("Request mapped and sending the jsp
file..."+name);
        return "home";
    }
```

NOTE : Here the client data parameter will match with the method parameter

name. If the parameters names are not matching then it will not assign to the variable.

4. To avoid it we use the annotation called as RequestParam to specify the name of the parameters.
   a. **Syntax** :

      @RequestParam(“urlParameter”)

      Example :

```
public String home(@RequestParam("user") String name,
HttpSession session)
```

5. When we are working on the Spring generally we don’t deal with the session hence we change to the ModelandView.
   a. **Syntax** :

```
@RequestMapping( “url”)
Public ModelAndView methodName( parameter ){
        ModelAndView modelAndViewObject = new
ModelAndView( );

        Return modelAndViewObject;
}
```

   Example :

```
@RequestMapping("/home")
public ModelAndView home(@RequestParam("user") String name) {

        ModelAndView modelAndView = new ModelAndView();
        //need to add the object and set the viewName
return modelAndView;
}
```

6. We add the data here with the help of addObject method.
   a. **Syntax** :

      modelAndObject.addObject(key, value);

      Example :

```
modelAndView.addObject("user", name);
```

7. We also need to set the view to display the response.
   a. **Syntax** :

      modelAndObject.setViewName(“jspName”);

      Example :

```
modelAndView.setViewName("home");
```

NOTE : For 1 value it is able to send the value but when you are passing different parameters we have to accept the parameters. If it is the case then we go with the object of a class.

Syntax :

```
@RequestMapping(“url”)
Public ModelAndView methodName( ClassName classObject ){
        ModelAndView modelAndViewObject = new ModelAndView(
);
        modelAndViewObject.addObject( key, object );
        modelAndViewObject.setViewName(“jspFileName”);
        return modelAndViewObject;
}
```

NOTE : To send the data from the page instead of the url we use the certain

request action.

Form Example :

```
	<p>
		<h1>Login Form</h1>
		<form action = "login">
			<label for = "userName">UserId : </label>
			<input type = "text" id = "userId" name = "id">
			<label for = "userName">UserName : </label>
			<input type = "text" id = "userName" name = "name">
			<input type ="submit">
			<input type = "reset">
		</form>
	</p>

	${user.getId()}
	${user.getName()}
```

HomeController :

```
	@RequestMapping("/home")
	public ModelAndView Index() {
		ModelAndView modelAndView = new ModelAndView();
		modelAndView.setViewName("home");
		return modelAndView;
	}

	@RequestMapping("login")
	public ModelAndView Index(Wizards wizard) {

		ModelAndView modelAndView = new ModelAndView();
		modelAndView.addObject("user", wizard);
		modelAndView.setViewName("home");
	return modelAndView;
    }
```

NOTE : If you want to use the save the data and don't want to use the view then go with the ModelObject. The model Object will be provided by the spring framework.

➔ If you are using the modelObject instead of the modeView then we cannot use the addObject( ) instead of it we use the addAtrribute.

Syntax :

@RequestMapping("url")
Public String methodName(parameter1 obj1, .., Model modelObject){
modelObject.addAttribute(key, value);
Return "jspFileName";
}

TIP : If you want to map the model then go with the modelMap.

Example :

```
	@RequestMapping("login")
	public String Index(Wizards wizard, ModelMap m) {
//		ModelAndView modelAndView = new ModelAndView();
		System.out.println(wizard.getId()+" "+wizard.getName());
```

```
                m.addAttribute("user", wizard);
//              modelAndView.setViewName("welcome");
        return "welcome";
    }
```

Tip : We are having annotation called as ModelAttribute. When you are using the ModelAttribute there is no requirement for us to use the ModelMap

Syntax :

@RequestMapping(“url”)
Public String Index(@ModelAttribute(“key” (optional)) className object){
Return “jspFileName”;
}

Example :

```
        @RequestMapping("login")
        public String Index(@ModelAttribute("user") Wizards user) {
//              ModelAndView modelAndView = new ModelAndView();
                System.out.println(user.getId()+" "+user.getName());
//              m.addAttribute("user", wizard);
//              modelAndView.setViewName("welcome");
        return "welcome";
    }
```

POST Method :

We can change the type from GET to POST with the help of method attribute in the forms. Also we can mention how the controller can handle the request based on the method type.

Syntax :

@RequestMapping(value = “value”, method = RequestMethod.POST)

NOTE: It will only work on the type of method which was mentioned there. Instead of writing the entire thing we can simply use the annotations.

1. For post method we can use the @PostMapping(value = “value”)
   a. **Example** :

```
        @PostMapping("login")
        public ModelAndView index(@ModelAttribute("user") Wizards user)
{
                ModelAndView mv = new ModelAndView();
                mv.addObject("user", user);
                mv.setViewName("welcome");
                return mv;
        }
```

2. For Get method we can use the @GetMapping(value = “value”)
   a. **Example** :

```
        @GetMapping("getData")
        public ModelAndView getData() {
                ModelAndView mv = new ModelAndView();
                List<Wizards> wizard = Arrays.asList(new Wizards(1,"Harry
Potter"), new Wizards(2, "Gilbert"));
                mv.addObject("Wizard", wizard);
                mv.setViewName("getData");
                return mv;
        }
```

Spring ORM :

Here we are going to map the database tables with the java class. To do that we use the ORM Frameworks like Hibernate and JPA.

1. **Configuring the Spring for ORM :**
    a. Create a spring starter app and select the web and also select the mysql driver, spring jpa as well.
    b. Create a controller
    c. Create a class to map with the data base.
    d. We need to enable the auto configuration feature. So that the spring boot can connect with the database.
        i. To do that simply go to the application.properties and use the below properties.

Syntax :

```
spring.application.name=SpringORMTrail

spring.datasource.url=jdbc:mysql://localhost:portNumber/databaseName
spring.datasource.username=userName
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=create

## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
```

Example :

```
spring.application.name=SpringORMTrail

spring.datasource.url=jdbc:mysql://localhost:3306/trail
spring.datasource.username=root
spring.datasource.password=0000
spring.jpa.hibernate.ddl-auto=create

## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
```

    e. We need to create a interface which extends the interface curd repository.
        i. **Syntax :**
            Interface interfaceName extends CurdRepository< className, Integer>

Example :

```
        public interface RepoClass extends CrudRepository<Student, Integer>
{

}
```

NOTE : The curdRepository Contains all the CURD operations at all.

f. Create a controller and get the data also create a object for the interface and use the annotation autowired so that the spring will provide the bean automatically.
   i. **Syntax** :

```
@Autowired
Interface InterfaceObject;
```

Example :

```
        @Autowired
        RepoClass repo;
```

g. Once you read the data in controller use the interface object and perform CURD operations using it.

Example .jsp file :

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
        <h1>Login</h1>
        <form action = "save">
                <input type = "text" name = "id" placeholder="Enter your id">
                <input type ="text" name = "name" placeholder="Enter your name">
                <input type = "submit" value = "save">
        </form>
                <form action = "find">
                <input type = "text" name = "id" placeholder="Enter your id">
                <input type = "submit" value = "find">
        </form>
        ${id }${name }
        ${student }
</body>
</html>
```

Example controller to saveData/ updateData :

```
@RequestMapping("/save")
        public ModelAndView Save(@ModelAttribute("id") int id,
@ModelAttribute("name") String name) {
                ModelAndView mv = new ModelAndView();
                System.out.println("Welcome to login page....");
                System.out.println(id+" "+name);
                mv.addObject("id", id);
                mv.addObject("name", name);
                Student student = new Student();
                student.setId(id);
                student.setName(name);
                repo.save(student);
                mv.setViewName("home.jsp");
                return mv;
        }
```

Example controller to find Data :

```
@RequestMapping("/find")
        public ModelAndView Find(@ModelAttribute("id") int id) {
                ModelAndView mv = new ModelAndView();
                Optional<Student> student = repo.findById(id);

                mv.addObject("student", student);
                mv.setViewName("home.jsp");
                return mv;
        }
```

Example controller to delete the data :

```
        @RequestMapping("/delete")
    public ModelAndView Delete(@ModelAttribute("id") int id) {
                ModelAndView mv = new ModelAndView();
                repo.deleteById(id);
                mv.setViewName("home.jsp");
                return mv;
        }
```

NOTE : If it is not able to create the beans then use the below annotations to work with

```
@EnableJpaRepositories(basePackages = "com.example")
@EntityScan(basePackages = "com.example")
```

Customizing the Queries :

We can write our own methods to fetch data using different attributes. All we need to do is to create a abstract methods in crudRepo

Syntax :

```
        Interface interfaceName extends CRUDRepository<className>{
                //Abstract methods
        }
```

Example :

```
public interface customerRepo extends CrudRepository<Customer, Integer>{
        List<Customer> findByName(String name);
        List<Customer> findByUserName(String name);
}
```

Example to fetch the data :

```
        @RequestMapping(value = "/FetchDataById")
        public ModelAndView FetchDataById(
                        @ModelAttribute("id")
                        int id) {
                ModelAndView mv = new ModelAndView();
                Optional<Customer> customer = cr.findById(id); //default method
                System.out.println(customer);
                mv.setViewName("GetData.html");
                return mv;
        }
```

```
	@RequestMapping(value = "/FetchDataByName")
	public ModelAndView FetchDataByName(
			@ModelAttribute("name")
			String name) {
		ModelAndView mv = new ModelAndView();
		System.err.println(cr.findByName(name)); //abstract method that we
created
		mv.setViewName("GetData.html");
		return mv;
	}

	@RequestMapping(value = "/FetchDataByUserName")
	public ModelAndView FetchDataByUserName(
			@ModelAttribute("user")
			String user) {
		ModelAndView mv = new ModelAndView();
		System.err.println(cr.findByUserName(user)); //abstract method that
we created
		mv.setViewName("GetData.html");
		return mv;
	}
```

HTML Code :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
	<form action="FetchDataById">
		<label for = "id">Id : </label>
		<input id ="id" name ="id" type="number" placeholder = "Enter the
id which you are looking for">
		<input type="submit" value ="get">
	</form>

	<form action="FetchDataByName" method="get">
		<label for = "name">Name : </label>
		<input id ="name" name ="name" type="text" placeholder = "Enter
the Name which you are looking for">
		<input type="submit" value ="get">
	</form>

	<form action="FetchDataByUserName" method="get">
		<label for = "user">userName : </label>
		<input id ="user" name ="user" type="Text" placeholder = "Enter the
UserName which you are looking for">
		<input type="submit" value ="get">
	</form>
</body>
</html>
```

NOTE : When you are creating the methods either user the findBy or getBy.

Always ends with the property Name or the column. We can even use the GreaterThan or lessthan in the methods.

Creating custom queries :

To create a custom queries we are having the HQL or JPQL. To use them we have to use the Query Annotation.

Syntax :

```
	@Query("hql/ jpql")
	abstractMethodName();
```

Example :

```
	@Query("FROM Customer where name = ?1 ORDER BY DateOfBirth")
	List<Customer> findByNameSorted(String name);
```

Representational State Transfer(REST):

REST is also known as RESTFUL parameters, which are used to send data between client and server.

Syntax :

```
	@RequestMapping("/URL/{wildcard}")
	Public returnType MethodName(@PathVariable("varName") parameters){
		//statements
	}
```

Example :

```
	@RequestMapping("/fetchName/{id}")
	@ResponseBody
	public String fetchName(@PathVariable("id") int id) {
		return cr.findById(id).toString();
	}
```

Return Data in form of json :

To return data in form of json you don't need to do anything as by default when you are fetching the data the spring will convert it to the json format and all you need to do is to not to convert the information to string that it.

Example :

```
	@RequestMapping("/fetchName")
	@ResponseBody
	public Iterable<Customer> fetch() {
		return cr.findAll();
	}
```

To pass the data or to manipulate it we have use the client support like postman. In postman we can find the history of the requests we are working one. By default it will only send the data in json format if we need the xml format then we have add a dependency for the xml which is Jackson Dataformat XML.

NOTE : We can even restrict what type of data format has to be produced.

Syntax :

```
	@RequestMapping( path ="url", produces={"application/format"})
	methodName(){
		//body of the method
	}
```

Example :

```
	@RequestMapping(path="/fetchName/{id}", produces =
{"application/xml"})
	@ResponseBody
	public Optional<Customer> fetchName(@PathVariable("id") int id) {
		return cr.findById(id);
	}
```

Sending Data to the server using the post method :
Here we are going to use the post method which will not show the data in the url.
Steps :
1. The major step is to declare the controller to RestController once you done that then you are creating the restful apis.
   a. **Syntax** :
```
@RestController
Public class className{
	//variables, Objects and methods
}
```
2. Replace the RequestMapping with the PostMapping annotation.
   a. **Syntax** :
```
@PostMapping('URL')
Method(){
	//action
}
```
   Example :
```
	@PostMapping("/save")
	public Customer CustomerSave(Customer customer) {
		cr.save(customer);
		return customer;
	}
```
3. We can even use the GetMapping annotation for the get method.
4. Also we can specify what type of data can be consumed and produced through API.
   a. *Consumes – Accepting of data*
   b. *Produce – Generating the data*

   Syntax :
```
@PostMapping(path="value", consumes ={"application/json or
application/xml"})
methodName(){
	//action
}
```
Put and Delete in REST :
1. We use the annotations for the put and delete methods.
   a. **Syntax** :
```
@PutMapping(path="url")
methodName(){
	//body of the method
}

@DeleteMapping(path="URL")
methodName(){
	//body of the method
```

```
                    }
```

Example :

```
        @DeleteMapping("/delete/{id}")
        public String DeleteCustomer(@PathVariable("id") int id) {
                cr.deleteById(id);
                return "Deleted";
        }
```

```
@PutMapping(path="/update")
        public Customer updateCustomer(Customer customer) {
                cr.save(customer);

                return customer;
        }
```

Spring Data REST :

With the help Spring Data REST we can ignore the controller. Here we need the following dependencies when creating the project.

1. *Database (H2/ MySQL Driver/...)*
2. *JPA*
3. *REST Repositories*

We need to use the annotation here to make it controller free that annotation is called as RepositoryRestResource.

Syntax :

```
        @RepositoryRestResource(CollectionResourceRel=”connectionName”,
path=”same as connectionName”)
                Public Interface RepoClassName extends
JPARepository<EntityName, Integer>
```

Example :

```
        @RepositoryRestResource(collectionResourceRel = "customer",
path="customer")
public interface CustomerRepo extends JpaRepository<Customer, Integer>{

}
```

Entity Class :

```
@Entity
@Table(name="customer")
public class Customer {
        @Id
        @GeneratedValue(strategy = GenerationType.SEQUENCE)
        private int id;
        private String name;
        private String DateOfBirth;
        private long phoneNumber;
        private String userName;

        public Customer() {
        }
        public Customer(int id, String name, String dateOfBirth, long
phoneNumber, String userName) {
                this.id = id;
```

```
		this.name = name;
		DateOfBirth = dateOfBirth;
		this.phoneNumber = phoneNumber;
		this.userName = userName;
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getDateOfBirth() {
		return DateOfBirth;
	}
	public void setDateOfBirth(String dateOfBirth) {
		DateOfBirth = dateOfBirth;
	}
	public long getPhoneNumber() {
		return phoneNumber;
	}
	public void setPhoneNumber(long phoneNumber) {
		this.phoneNumber = phoneNumber;
	}
	public String getUserName() {
		return userName;
	}
	public void setUserName(String userName) {
		this.userName = userName;
	}
	@Override
	public String toString() {
		return "Customer [id=" + id + ", name=" + name + ", DateOfBirth=" +
DateOfBirth + ", phoneNumber=" + phoneNumber
				+ ", userName=" + userName + "]";
	}


}
```

MainMethod :

```
@SpringBootApplication
@EntityScan(basePackages = "com.example")
public class AxisBank1Application {

	public static void main(String[] args) {
		SpringApplication.run(AxisBank1Application.class, args);
	}
}
```

Properties :

```
spring.application.name=AxisBank-1

spring.datasource.url=jdbc:mysql://localhost:3306/axisbank
spring.datasource.username=root
spring.datasource.password=0000
spring.jpa.hibernate.ddl-auto=update


## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
```

Perquisites :

1. Tomcat Server
   a. Adding the server into the eclipse :
      i. Click on window
      ii. Click on the properties.
      iii. Click on the add server -> select the server type and version.
      iv. Click on finish

Steps to create a project :

1. We need to create a dynamic web project (check the web.xml generation)
2. We need to change the java version.

We need to get the dispatcher-servlet

NOTE :

⇨ It is best practice to write the database related objects/ files as DAO in repository Package .
Syntax :
    Interface EntityNameDAO

⇨ If any we are implementing the interface then we have to name the className end as DAOImpl
Syntax :
    Class ClassNameDAOImpl
        Or
    Class DefaultClassNameDAP (single implementation)

⇨ For Service It is suggested to use the Service at the end of the interface