**All ABOUT SPRINGBOOT**

Author: Sainath Muthyala (#**2520910**)

**Table of Contents**

[Section -1 about Spring Boot 2](#_Toc145326180)

[Section -2 Importance of Dependency Injection 3](#_Toc145326181)

[Section -3 Simple Springboot Application to understand the Flow 7](#_Toc145326182)

[Section -4 Bean Scope 10](#_Toc145326183)

[Section -5 Autowired 11](#_Toc145326184)

[Section-6 Developing Web Application using SpringBoot 14](#_Toc145326185)

[Section -7 About Dispatcher Servlet 15](#_Toc145326186)

[Section -8 Spring Boot MVC using JPA to connect to H2 Database 18](#_Toc145326187)

[Section -9 SpringBoot Application using customized h2 DB queries 22](#_Toc145326188)

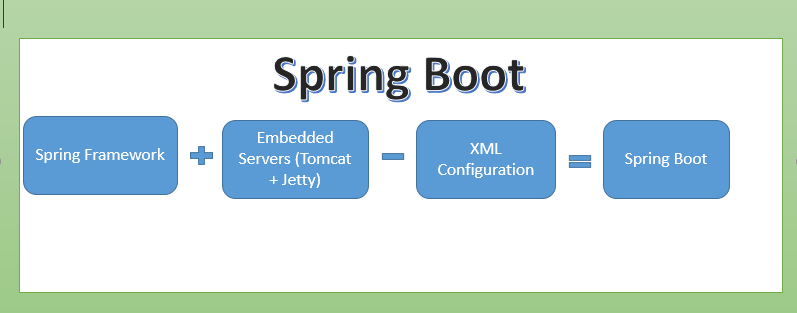
[Section -10 Service Interface 24](#_Toc145326189)

[Section -11 MicroService Architecture 29](#_Toc145326190)

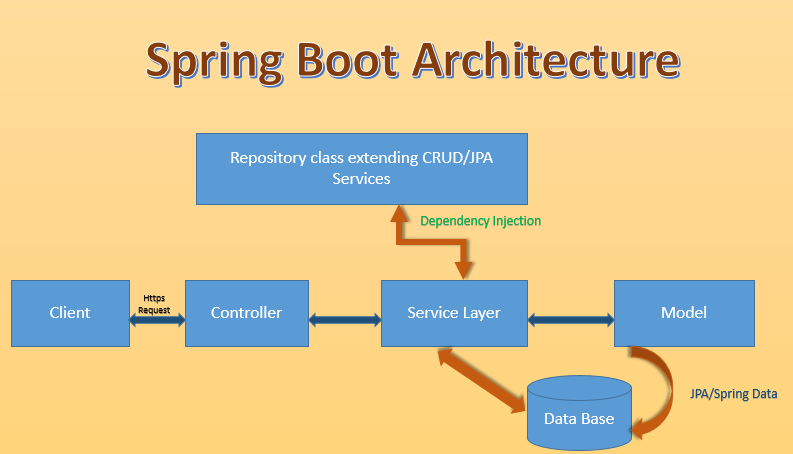
[Section -12 Few Enterprises, which are using Spring Boot 30](#_Toc145326191)

# Section -1 about Spring Boot

* Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.
* Spring Boot is a Framework which is developed to focus on the coding part rather than configuration. Unlike Spring Framework where we need to add the configuration and external web server (Tomcat/Glassfish) and also we need to deploy on the webserver every time whenever we want to test the application during the development phase.
* Spring Boot uses completely new development model to make Java Development very easy by avoiding some tedious development steps and boilerplate code and configuration.



* Spring Boot Framework Programming model is inspired by Groovy Programming model. Spring Boot internally uses some Groovy based techniques and tools to provide default imports and configuration. Spring Boot Framework also combines existing Spring Framework annotations into some simple or single annotations.



**Spring Boot Architecture**

The above diagram tells about the Architecture of the Spring Boot framework. We will discuss more about the same in this document.

# Section -2 Importance of Dependency Injection

**Dependency Injection (or) Inversion of Control**: It tells “**Not to create or initiate an object inside another java class, instead rely on spring’s IOC to create the object for you**”.

**Example:**

Class **A**{

Function **A**{

**B** objectB = new **B**();

//**Note**: DI or IOC will not encourage to create the objects inside another class

}

}

🡪 Here, Class A is completely dependent on Class B

🡪If Class B is modified or replaced by Class C, then it may force you to modify

Class A as well.

Here comes, DI or IOC containers importance. This module will help in overcoming in all the above mentioned dependency problems.

**Understanding Dependency Injection or IOC**

**Problems if you do not use DI**:

The below example will let you know the problems of not using DI.

Class **Training** {

Public void ConductTraining(){ //ConductTraining is a method inside Training class.

//code to conduct training

            }

}

Class **MyProject**

{

**Training** obj= new **Training** (); //Creating the “Training” class object using new operator inside **MyProject**.

Public void ConductJavaTraining {          //ConductJavaTraining is a method inside **MyProject**.

 obj. ConductTraining();

}

   Here, MyProject class is dependent on Training class. Training class object is created in MyProject class using new operator and then “ConductTraining” method is called using the object inside ConductJavaTraining method of MyProject class.

The below are the problems, which will occur from the above scenario.

**Problem 1**: If “Training” class is replaced with another class say “KnowledgeSharing”, then we need to have object of “KnowledgeSharing” class, and we need to make necessary changes in the MyProject class as well. If the application is a large one and has lot of dependency on ‘‘Training” class then it make take lot of Developer’s time to make necessary changes.

**Problem 2**: If Training class is modified say parameterizing its constructor, then the MyProject class or the application will surely needs to be modified accordingly.

**Problem 3**: You cannot test the “MyProject” class without including the “Training” class.

All the above problems are only for one reason –“ MyProject class is directly dependent on Training class”. In other words, MyProject class is “**Tightly coupled**” with Training class.

**Understanding Dependency Injection or IOC**:

Concept Explanation:

Instead of creating dependency objects, we can outsource the task of handling all dependencies of an object to an **expert** (**some third party program written by you or preexisting one like spring framework**).

In order to do this, you need an Interface.

**Example**: Interface Training {

Public void ConductTraining();

} //end of interface Training

Class JavaTraining implements Training

{

@override

Public void ConductTraining ()

// we should override the method of ConductTraining Interface

{

//code to conduct Training

}

}//end of class JavaTraining

Class MyProject

{

Training training;

MyProject (Training training) //**Constructor method way**

{

This. training = training;

}

**(OR)**

Public void setMyTraining (Training training) //**Setter method way**

{

This. training = training;

}

Public void PrepareForTraining

{

training. ConductTraining();

}

Spring DI framework provides two ways of DI:

1. **Using the constructor way**
2. **Using the Setter method way**

1. **Constructor way**: Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.

2. **Setter method way**: Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

**Note 1**: All the configuration changes has to be mentioned in the spring configuration file.

**Note 2**: You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

# Section -3 Simple Springboot Application to understand the Flow

**Program-1**

Package com.xxxx.xxx

Import org.springframework.boot.SpringApplication

@SpringBootApplication

Public class SimpleSpringbootApplication {

Public static void main (String [] args ) {

SpringApplication.run(SimpleSpringbootApplication.class args);

}

}

We can run this application in Spring Tool Suite(STS) by using “Spring Tool App” in “Run As” method.

**@SpringBootApplication** 🡪 This tells the compiler to load all the auto-configuraitons ,methods and component scanning.If we are using Spring framework then we need to declare a class with @configuration,@EnableAutoConfiguration and @ComponentScan annonations and @ SpringBootApplication is equivalent to all the above annotations. This is how most of the boilerplate code and configurations are reduced in SpringBoot.

**Program-2** 🡪

Package com.xxx.xxx

Import statements //add all the import statements as required.

**@component** // To Inform sprint container to create object which can be used by other classes.

Public class Employee

{

Private int eid;

Private String ename;

Private String etech; //below are the setters and getter methods

Public int getEid(){

return eid;

}

Public void setteid (int eid) {

This.eid =eid;}

Public String getename() {

return ename;

}

Public void settename (string ename) {

This.ename =ename;}

Public String getetech() {

return etech;

}

Public void setetech (string etech) {

This.etech =etech;}

Public void PrintTxt() // **This method is used to help us to print the text when called in Program-1**

{

System.out.println (“Welcome to IT World”)

}

}

Note : In order to use program-2 in program-1 ,before Dependency Injection ,we used to create the object of program-2 in program-1, like shown below.

**Employee emp = new Employee ();** //Dependency Injection(DI) does not encourages to use new operator to create the object. Instead by using DI ,we can add program -2 in program-1 as below.

|  |
| --- |
| ConfigurableApplicationContect context = SpringApplication.run(SimpleSpringbootApplication.class args)  Employee emp = context.getBean(Employee.class)  Emp.PrintTxt();  //  **Output** : We will get an error saying that no qualifying bean object .  **Solution** :And in order to fix this error we need to use the annotation **@component** on top of the Employee classwhich will tell the sprint container to create the object and to inject the same when called by other program by using getbean method. After adding the annotation ,we will get the output as “Welcome to IT World”. |

# Section -4 Bean Scope

Lets add a constructor to program-2 to understand the Bean Scope. We have two types of Bean Scope 1) Singleton and 2) Prototype

Public Employee(){ //constructor in Employee class

Super();

System.out.println(“Hurry !!! Object created”);

}

If we run the application i.e.Program-1 ,then we will get the output as “**Hurry !!! Object created**”

And if we comment the below lines also, we will get the above output and this is because by default sprint container will use the Singleton scope.

|  |
| --- |
| // Employee emp = context.getBean(Employee.class)  // Emp.PrintTxt(); |

In order to use Prototype scope ,then we need to use below annotation

@scope(value =”prototype”) ,now when ever you use .getBean method then those many time that particular class will be called.

Employee emp = context.getBean(Employee.class)

emp.PrintTxt();

Employee emp1 = context.getBean(Employee.class)

Emp1.PrintTxt();

**Output** : “**Hurry !!! Object created**”

“**Hurry !!! Object created**”

**Note** : If you comment above lines ,then no object will be created and you will not see the line “**Hurry !!! Object created**” .

# Section -5 Autowired

In order to understand the concept of Autowiring ,lets create a new class Laptop as program-3.

**Program-3**

Package com.xxxxx.xxxx

@component

Public class laptop{

Private int lid;

Private string brand;

//generate setters and getters

Public int getLid() {

return lid;

}

Public void setLid(int lid){

this.lid = lid;

}

Public String getBrand() {

return brand;

}

Public void setBrand(String brand){

this.brand = brand;

}

@override

Public String toString() {

return “Laptop [lid=’’ + lid + “ , brand=” +brand+ ‘’]”;

}

Public void compile() {

System.out.println(“compiling…”);

}

}

Now ,in order to inject Laptop class in Employee class (program-2),we need to add the Laptop variable and the getter and setter methods of Laptop class in program-2.

@**autowired** //to bind the laptop object in Employee class

Private Laptop laptop; // Declare the laptop variable in Employee class

Public Laptop getLaptop() {

return laptop;

}

Public void setLaptop(Laptop laptop) {

this.laptop = laptop;

}

Public void PrintTxt() // **This method is used to help us to print the text when called in Program-1**

{

System.out.println (“Welcome to IT World”);

laptop.compile();

}

**Note** : Inorder to create Laptop object ,we need to add annotation @component on top of Laptop class. Although we have created two object one for Employee and the other for Laptop class by using @component ,but how Employee class know to inject Laptop class as Employee class has dependency on Laptop class and this is where sprint container will bind them by using autowire. Add @**autowired** on the laptop variable which is declared in Employee class.

**Output** : you will get the output as “compiling…”

Note : If you want to give some name to the Laptop class ,then add the name at the annontation @component

Example **@component(“Lap1””)**

And if you want to search by name in Employee class ,then we should use **@qualifier** annotation below @autowired annotation i.e.,

@autowired

@qualifier(“lap1)

**Autowiring Modes** :

Autowiring in spring has 5 different modes.

1. **No**            :  This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring.
2. **byName**           :  Autowiring by Property Name .  Spring container looks at the properties of the beans on which autowire attribute is set tobyName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
3. **byType**             :  Autowiring by property datatype. Allows a property to be autowired if exactly one bean of the property type exists in the container.
4. **constructor**   :  Analogous to byType, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
5. **Autodetect** : Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType.

**Note**: You can use byType or constructor autowiring mode to wire arrays and other typed-collections.

**Limitation of Autowiring :**

1. Explicit dependencies in property and constructor-arg settings always override autowiring.
2. Autowiring is less exact than explicit wiring.
3. Wiring information may not be available to tools that may generate documentation from a Spring container.
4. You cannot autowire so-called simple properties such as primitives, Strings, and Classes.
5. If no unique bean definition is available, an exception is thrown.

# Section-6 Developing Web Application using SpringBoot

In order to create SpringBoot Web MVC application select web option in the STS IDE.

1)Create a folder with name “Webapp” in the “main” folder and create a jsp file with name home.jsp.

**Note** :SpringBoot by default doesnot support JSP ,and if we are using JSP ,then we need to add the JSP dependencies in pom.xml file.

**<**dependency>

**<**groupId>org.apache.tomcat</groupId>

**<**artifactId>tomcat-jasper</artifactId>

<version>8.x.x </version> 🡪 We need to select the version based on the tomcat version which is part of SprintBoot.

2) Create a Controller class which handles the requests.

**Note** : We need to use **@controller** annotation to let spring container know that the particular class will acts as controller to handle the request and will mapping the request to respective methods based on the annotation **@RequestMapping** annotation.

**Sample controller program using HttpSession**

Import javax.servelet.http.HttpSession; // to import all the classes and methods of HttpSession

Import org.springframework.stereotype.Controller;

//to import all the classes and methods of Controller

Import org.springframework.web.bind.annotation.RequestMapping;

Import org.springframework.web.bind.annotation.RequestParam;

//to import all the classes and methods of RequestMapping and RequestParam

@Controller

Public Class OrgController {

@RequestMapping (“Org”) // This will check the request and will call the respective method

Public String Org(@RequestParam(“name”) String myName, HttpSession session)

{

System.out.println(“hi”+ myName);

Session.setAttribute(“name” ,myName);

return “home”;

}

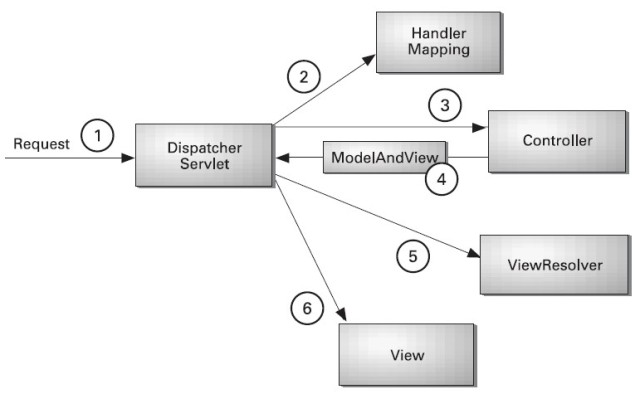
}

**Sample controller program using ModelAndView Object**

**Note** : ModelAndView is a special class ,where it can handle data(model) and the view. This Internal uses Dispatcher Servlet which handles the flow.

# Section -7 About Dispatcher Servlet

**Understanding the flow of Spring Web MVC:**



1. All the incoming requests are intercepted by the DispatcherServlet that works as the “front controller (which is one of the design patterns)”.
2. In Spring MVC ,the DispatcherServlet gets entry of handler mapping from the xml file.And this is an inbuilt controller in Springboot ,hence I will not look for xml file.
3. Once getting entries from handler mapping, it forwards the request to the controller.
4. The controller returns an object of ModelAndView.
5. The DispatcherServlet checks the entry of view resolver in the xml file. Here in Spring Boot ,it checks the Application.properties file for the path of the views.
6. And finally it invokes the specified view component.

Import javax.servelet.http.HttpSession;

Import org.springframework.stereotype.Controller;

Import org.springframework.web.bind.annotation.RequestMapping;

Import org.springframework.web.bind.annotation.RequestParam;

@Controller

Public Class OrgController {

@RequestMapping (“Org”) // This will check the request and will call the respective method

Public String ModelAndView Org(@RequestParam(“name”) String myName, HttpSession session)

{

ModelAndView mv = new ModelAndView ();

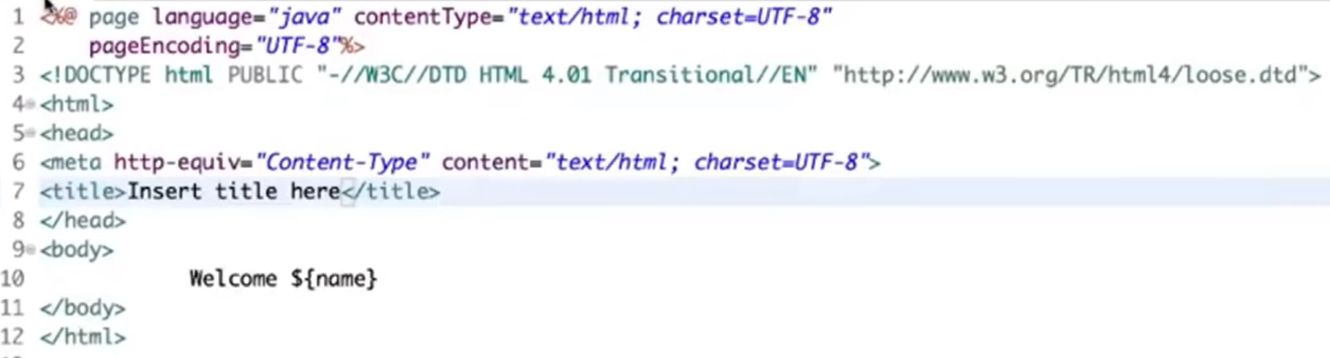
mv.addObject(“name” ,myName);

mv.setViewName(“home”); //Here we are setting view name ,home.jsp is the view here

return mv;

}

}

**home.jsp page**

In browser, If you give the URL localhost:8080/org?name=Sainath ,then if we run the above controller class the output will be

**Output** : Welcome Sainath

**Sample controller program using ModelAndView Object and by passing Pojo class a parameter**

Lets say ,if we are passing three object in the browser as shown below.

We already have program-2 which is Employee.java class which has the variables eid,ename and etech ,lets use the same to capture the data which is coming from browser.

In the browser, if you are sending three values

Localhost:8080/home?eid==29&aname=Sainath&etech=java

Lets see how the data is handled by controller class i.e.,

@Controller

Public Class OrgController {

@RequestMapping (“Org”) // This will check the request and will call the respective method

Public ModelAndView Org(Employee employee)

{

ModelAndView mv = new ModelAndView ();

mv.addObject(“obj” ,employee);

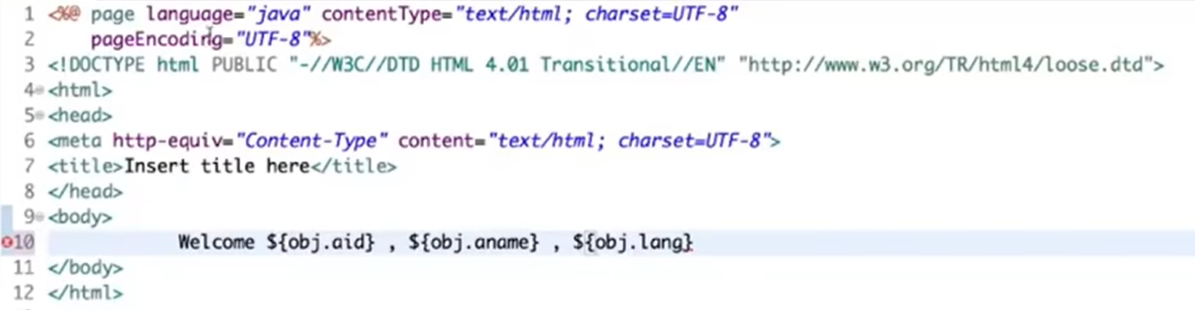
mv.setViewName(“home”); //Here we are setting view name ,home.jsp is the view here

return mv;

}

}.

We need to add the below objects using jstl tags in the home.jsp page. Here home.jsp page acts has view,Employee.java class acts as model class and OrgController.java acts as controller class.



**Output** : Welcome 29 ,sainath ,java

**Application.Properties file**

Application. Properties file is used to add the configuration as per our need .Although Spring Boot does not encourages to concentrate more on the configuration part, however we have that flexibiity to add our own custom configurations using the Application.Properties file.

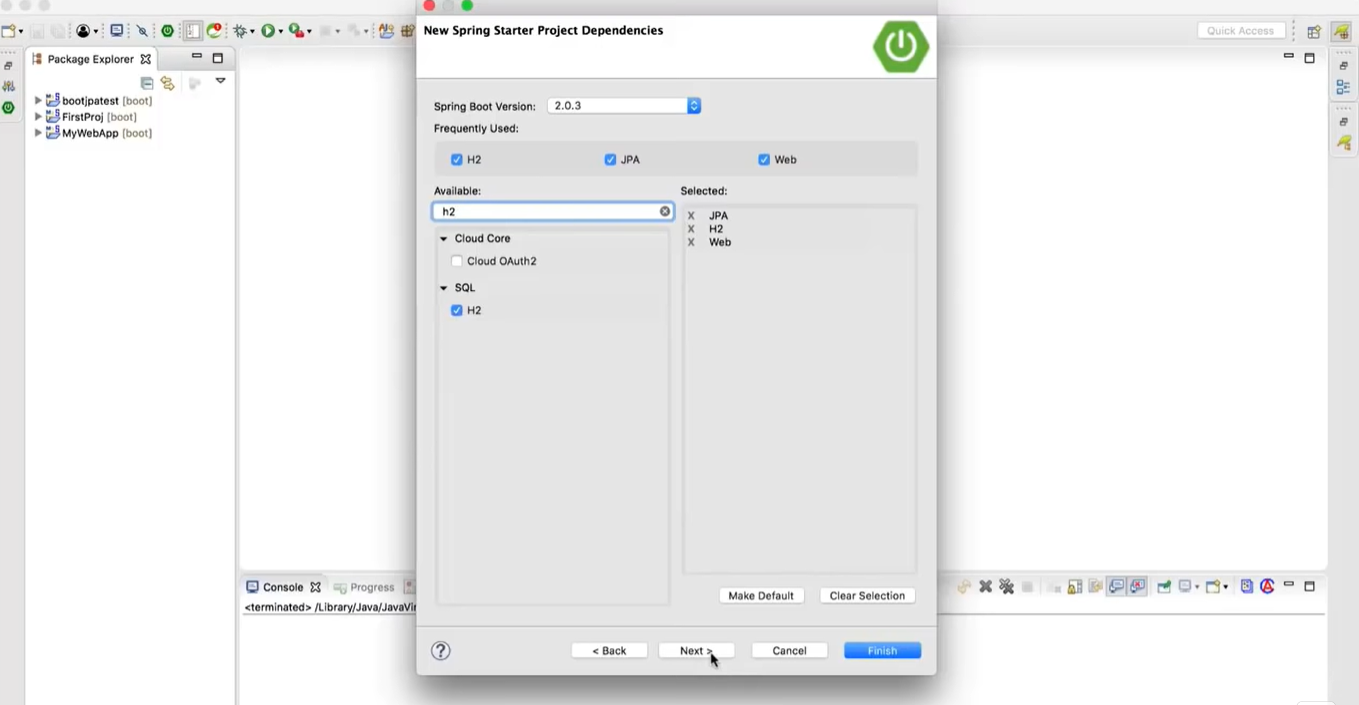
**Spring.mvc.view.prefix=/pages/** -> This tells where our view pages are available.

**Spring.mvc.view.suffix=.jsp -**> This tells which suffix pages has to be supported by the application ,here in this case it is .jsp files.

**Note:** If you have worked on Spring MVC , we have the concept of “view resolver” in which we will add the above things.

# Section -8 Spring Boot MVC using JPA to connect to H2 Database

In order to create application using ,we have to select “web” for webapplication JPA(Java Persistence API) to connect to DB and to perform CRUD operations and H2 Database which is an in memory DB .



Create a model package in the application and in the same ,create a model class i.e., pojo class with setters and getters methods. Lets use the same Employee.class as shown above.

**Employee.java**

Package com.xxx.xxx

Import statements //add all the import statements as required.

**@component** // To Inform sprint container to create object which can be used by other classes.

**@entity** // To inform that this is an entity class and can be used to set and get the data from DB.

Public class Employee

{

**@id**  //This tell that the id is a primary key and will be unique in the table

Private int eid;

Private String ename;

//below are the setters and getter methods

Public int getEid(){

return eid;

}

Public void setteid (int eid) {

This.eid =eid;}

Public String getename() {

return ename;

}

Public void settename (string ename) {

This.ename =ename;}

}

Public string toString () { // The toString() method returns the String representation of the object.

Return “Employee [eid==” +eid+ ” ,ename=” + ename+ “]”;

}

}

**home.jsp**

Create home.jsp file under webapp folder as shown below.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Login page in HTML</title>

</head>

<body>

<h1>Adding employee details</h1>

<form action="addEmployee">

<input type="text" name="eid"><br>

<input type="text" name="ename"><br>

<input type="submit"><br>

</form>

<form action="getEmployee">

<input type="text" name="eid"><br>

<input type="submit"><br>

</form>

</body>

</html>

**homeGet.jsp** // to pass the id in order to fetch the data from DB.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Login page in HTML</title>

</head>

<body>

<h1>Fetching employee details</h1>

<form action="getEmployee">

<input type="text" name="eid"><br>

<input type="submit"><br>

</form>

</body>

</html>

Lets create controller class to handle the request

@controller

Public class EmployeeController

{

@Autowired;

EmployeeRepo repo;

@RequestMapping(“/addEmployee”)

Public string addEmployee( Employee employee)

{

repo.save(employee); //to save the data in h2 database.

return “home.jsp”; //Once after saving the data ,page will once again return to home.jsp

}

@RequestMapping(“/getEmployee”)

Public ModelAndView getEmployee( @RequestParam int eid)

{

//here we are going to get id as input and the same has to be

ModelAndView mv = new ModelAnd View(“**showEmployee.jsp**”)

Employee employee = repo.findById(eid).orElse(new Employee); //to fetch the data from h2 database based on the id orElse is used in order to handle null value.

return “home.jsp”; //Once after saving the data ,page will once again return to home.jsp

}

} //showEmployee.jsp is used to display the data based on the the id which we are passing.

Note : Here we are passing Employee class which will capture the data coming from home.jsp file .

In order to use h2 database ,which is an inbuilt database ,we need to do some configurations in the **application.properties** file.

**Spring.h2.console.enabled =true**

**Spring.datasource.platform=h2**

**Spring.datasource.url=jdbc:h2:mem:MyDatabase**

In order to access the h2 database go to browser and type the below url

**URL: localhost8080/h2-console**

In general ,we have to create tables in the MyDatabase DB and then we need to map the bales to the pojo classes with the annotations @entity ,@id in the Employee.java class .These things will be taken care by SpringBoot only .

In order to perform CRUD operations in h2 database ,let create a new interface as shown below.

Import org.springframework.data.repository.CrudRepository;

Public interface EmployeeRepo extends **CrudRepository** (Employee ,Integer)

{

}

//here we need to pass the class name which is going to be used to perform

CRUD operation and the primary key type ,here in Employee.java class primary key is id and is of Integer type.

Note: This interface has to be added in controller class by using @Autowired annotation.

CrudRepository is an Interface which springBoot has implemented all the CRUD operations.

Sample records ,which are present in the database .

# Section -9 SpringBoot Application using customized h2 DB queries

Suppose if we want to fetch the records by using our own customized queries ,then the same can be done without much coding to write the query.

Here, in controller class

@controller

Public class EmployeeController

{

@Autowired;

EmployeeRepo repo;

@RequestMapping(“/addEmployee”)

Public string addEmployee( Employee employee)

{

repo.save(employee); //to save the data in h2 database.

return “home.jsp”; //Once after saving the data ,page will once again return to home.jsp

}

@RequestMapping(“/getEmployee”)

Public ModelAndView getEmployee( @RequestParam int eid)

{

//here we are going to get id as input and the same has to be

ModelAndView mv = new ModelAnd View(“**showEmployee.jsp**”)

Employee employee = repo.findById(eid).orElse(new Employee); //to fetch the data from h2 database based on the id orElse is used in order to handle null value.

System.out.println(**repo.findByTech(“java”**); //This is a custom method which will return all the records with the technology as java.

System.out.println(**repo.findByeidGreaterThan(“id”**);//to fetch the records greater than the id which we are passing.

System.out.println(**repo.findByTechSorted(“java”**);

return “home.jsp”; //Once after saving the data ,page will once again return to home.jsp

}

}

We need to add findByTech in the interface EmployeeRepo.

Import org.springframework.data.repository.CrudRepository;

Public interface EmployeeRepo extends **CrudRepository** (Employee ,Integer)

{

List <Employee> **findBy**Tech(string tech);

List <Employee> **findBy**EidGreaterThan(int eid); //This method will fetch all the records greater than the id which we are passing.

@Query(“from Alien where tech =?1 order by ename””)

List<Employee> findByTechSorted(string tech);

//here we are writing our own query using jpql query which is similar to hql query

}

Note: Here we can write our own custom queries which starts with “findBy” or “getBy” and which follows the variable which we declared in the pojo class.

**Important note** : Here we are not using any service layer since we are only discussing the concepts and the flow ,but if we need to create realtime enterprize application,then we need to go for service class.

In order to use service layer ,we should create a service Interface and the service Implementation class of the service Interface.

# Section -10 Service Interface

**EmployeeService.java**

package com.xxxx.xxxxx.xxxx;

import java.util.List;

public interface IEmployeeService { //here we should use the naming convention the service Interface name should start with ‘I’.

Public Employee addEmployee( Employee employee)

public List<Employee> getAllEmployees();

public Invoice getEmployeeById(int id);

public void deleteEmployeeById(int id);

public void updateEmployee(Employee employee);

}

**Implementation class**

**EmployeeServiceImpl.java**

package com.XXX.springboot.service.impl;

import java.util.List;

import java.util.Optional;

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

import org.springframework.stereotype.Service;

@Service

public class EmployeeServiceImpl implements IEmployeeService{

@Autowired

private EmployeeRepo repo;

@Override

public Employee saveEmployee(Employee employee) {

return repo.save(employee);

}

@Override

public List<Employee> getAllEmployees() {

return repo.findAll();

}

@Override

public Invoice getEmployeeById(int id) {

Optional<Employee> opt = repo.findById(id);

if(opt.isPresent()) {

return opt.get();

} else {

throw new InvoiceNotFoundException("Employee with Id : "+id+" Not Found");

}

}

@Override

public void deleteEmployeeById(int id) {

repo.delete(getEmployeeById(id));

}

@Override

public void updateEmployee(Employee employee) {

repo.save(employee);

}

}

**Controller Class**

package com.xxxx.xxxxx.xxxxxx;

import java.util.List;

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

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

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

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

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

import org.springframework.web.servlet.mvc.support.RedirectAttributes;

@Controller

@RequestMapping("/employee")

public class EmployeeController {

@Autowired

private IEmployeeService service;

@PostMapping(“/addEmployee”)

Public string addEmployee( Employee employee)

{

repo.save(employee); //to save the data in h2 database.

return “home.jsp”; //Once after saving the data ,page will once again return to home.jsp

}

}

@GetMapping(“/getEmployee”)

Public ModelAndView getEmployee( @RequestParam int eid)

{

//here we are going to get id as input and the same has to be

ModelAndView mv = new ModelAnd View(“showEmployee.jsp”)

return “home.jsp”;

}

@PostMapping("/update")

public String updateEmployee(@RequestParam int eid)

{

service.updateEmployee(employee);

int id = employee.getId();

System.out.println("Employee with id: '"+id+"' is updated successfully !");

return "home.jsp";

}

@GetMapping("/delete")

public String deleteInvoice(

@RequestParam int eid) {

try {

service.deleteEmployeeById(id);

System.out.println("Employee with Id : '"+id+"' is removed successfully!");

} catch (InvoiceNotFoundException e) {

e.printStackTrace();

}

return "home.jsp";

}

}

**Fetching the data in JsON format**

If we are using Restful Webservices ,then we need to get the data in Json/xml format and this is needed in order to communicate or integrate with other services.

Below are the changes which we need to do in order to go with Json format.

**Change 1**: We need to extend the JPA repository instead of CRUD repostitory by extending the same ,we will be getting additional methods which are not supported by CRUD repository.

Import org.springframework.data.repository.CrudRepository;

Public interface EmployeeRepo extends **JpaRepository** (Employee ,Integer)

{

}

**Change 2:**

@controller //we can use @Restcontroller for Restful service operations.

Public class EmployeeController

{

@Autowired;

EmployeeRepo repo;

}

@RequestMapping(“/Employee”)

@ResponseBody

Public List<Employee> getEmployee( )

{

return repo.findAll().toString();

}

@RequestMapping(“/getEmployee/{eid}”)

@ResponseBody

Public Optional<Employee> getEmployee( @Pathvariable(“eid”) int eid)

{

return repo.findById(eid); //to fetch the data from h2 database based on the id orElse is used in order to handle null value.

}

}

Output: This is how the output we can see in json format after running the program.



**Restricting the data to only xml format**

We can restrict the data and can be allowed by the services to fetch the data only in xml format then we need to add produces option in the request mapping as shown below .Please note by default spring restful service support json and xml formats.

@RequestMapping(path = “/Employee” ,produces ={“application/xml”})

@ResponseBody

Public List<Employee> getEmployee( )

{

return repo.findAll().toString();

}

We can use postman/Swagger tool to fetch the records(Get method),to save the records(post method),to update the records(put method) and to delete(delete method) the records from DB.

SpringBoot is a preffered language used for MicroService architecture.

# Section -11 MicroService Architecture

A microservices architecture allows teams deploy independent applications without affecting other services in the architecture.

**Few benefits of MicroService Architecture**

1: Improved productivity.

2: Better resiliency.

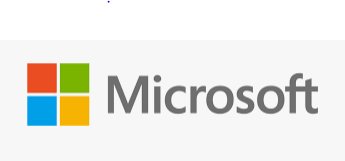
3: Increased scalability.

4: Continuous delivery/continuous integration.

5: Optimize business functionality.

# Section -12 Few Enterprises, which are using Spring Boot

**References**

* Online Trainings and Material