

# SPRING WEB MVC FRAMEWORK

Instructor:



1

- Introduction to Spring Web MVC

2

- Spring Controller

3

- Resolving Views

4

- Spring @Autowired

5

- Spring MVC First Example

## ❖ After the course, attendees will be able to:

---

Understand Spring Web MVC Framework and its core technologies.

---

Know how to write a Web application with Spring MVC Framework.

---

## Section 1

# INTRODUCTION

# Introduction to Spring MVC

A part of the **Spring Framework** is **Spring Web MVC**, an extensible MVC framework for creating web applications.

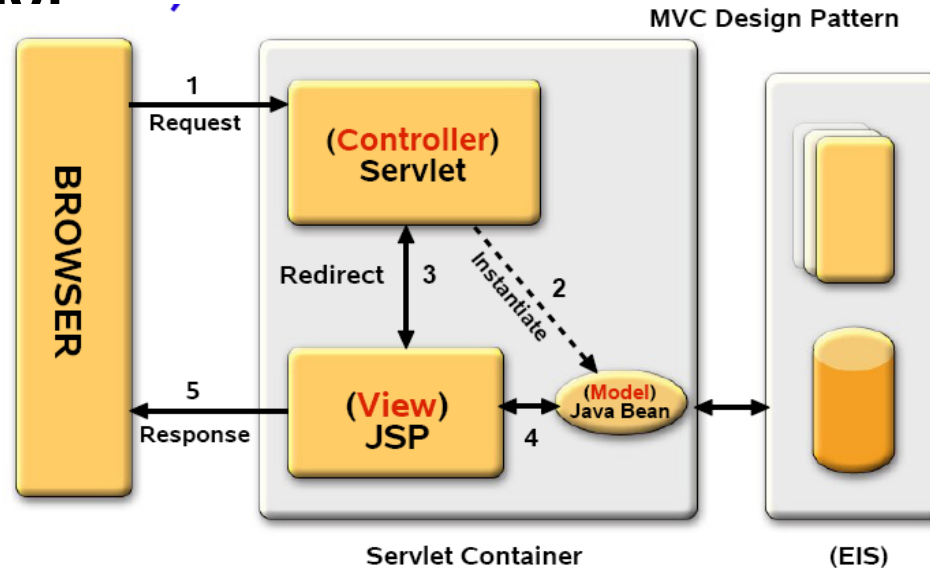
Support for Invesrion of **Control** (or, the Dependency Injection).

Use **DispatcherServlet** that dispatches request to handler.

The default handler is based on the **@Controller** and **@RequestMapping** annotations.



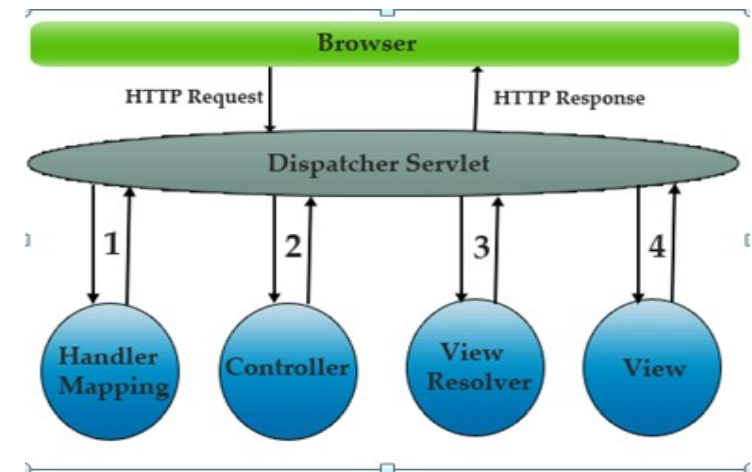
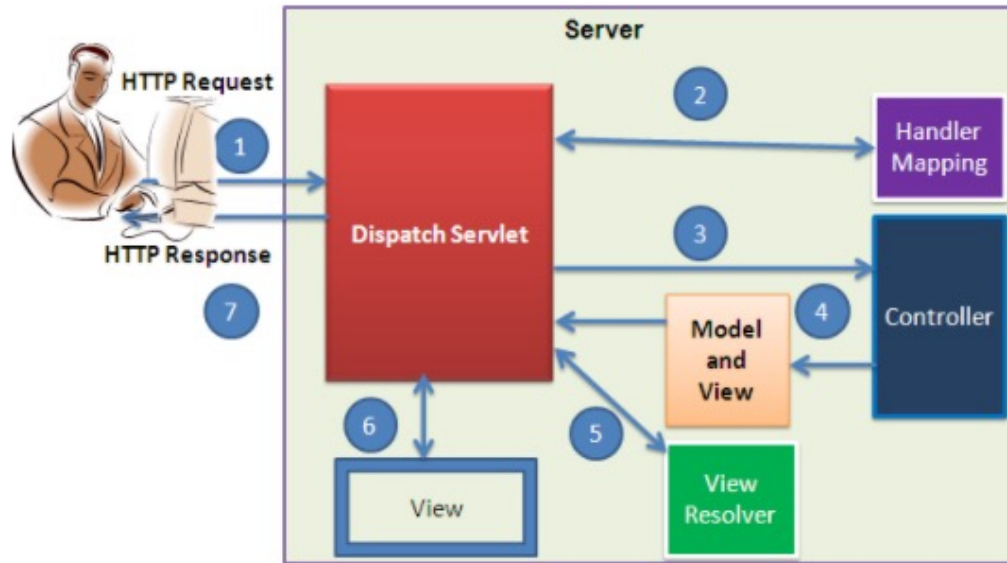
## ❖ A Web Application developed using JSP/Servlet Technology



- ❖ **M**odel : presents Data;
- ❖ **V**iew : presents UI part;
- ❖ **C**ontroller : manages the application flow, makes a call to some sort of service producing Model and then passes on the Model to the View;

# An Introduction to MVC

## ❖ A Web Application developed using SpringMVC



# Spring MVC Processing Sequence

## Step1

- Client sends **HTTP Request** from the web browser to server. Inside the Web or Application server, **Dispatch Servlet** will be there to handle your Http request and Processed HTTP Response.

## Step2

- **Handler Mapping** Maps incoming HTTP Requests to handlers, In spring MVC framework we use **@Request Mapping** annotation to map the incoming Http Request with Model and View Object inside the controller class.

## Step3

- Controller class in spring is implemented with the help of **@Controller** Annotation, which does model Map with data and extracting a view name, it can also send directly as HTTP response without mapping the incoming request with model data object.

## Step4

- The Controller class has **ModelAndView** object that has the model or Data and the view name, the Controller class executes the incoming request by calling the respective service method and returns ModelAndView Object to DispatcherServlet.

## Step5

- The DispatcherServlet will send the view name to a **ViewResolver** to find the original View (.jsp) page to display.

## Step6

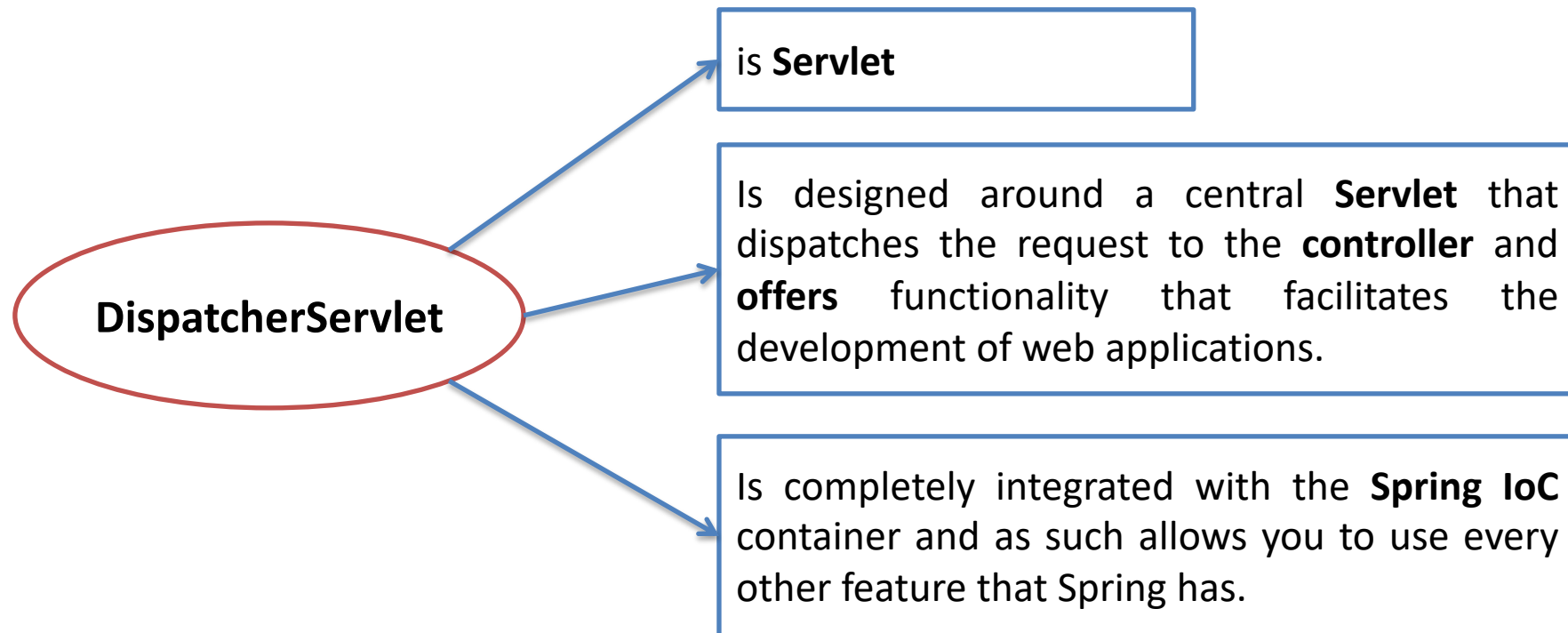
- The DispatcherServlet will then send the model object to **View** in order to render the result.

## Step7

- The view (.jsp) page will show the **HTTP Response** back to Client on the web browser.



## ❖ What is Spring DispatcherServlet?



- ✓ **DispatcherServlet** acts as front controller for Spring based web applications.
- ✓ It provides a mechanism for request processing where actual work is performed by **configurable**, **delegate components**.
- ✓ It is inherited from **javax.servlet.http.HttpServlet**, it is typically configured in the **web.xml** file.

## ❖ Configuration of **DispatcherServlet** in web.xml

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">
  <display-name>Archetype Created Web Application</display-name>
  <welcome-file-list>
  <welcome-file>/views/login.jsp</welcome-file>
  </welcome-file-list>

  <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

- ❖ **Spring container** creates objects and associations between objects, and manages their complete life cycle. These container objects are called **Spring-managed beans** (or simply beans), and the container is called an **application context** (via class **ApplicationContext**) in the Spring world.
- ❖ When DispatcherServlet is loaded, *it looks for the bean configuration file of WebApplicationContext and initializes it.*

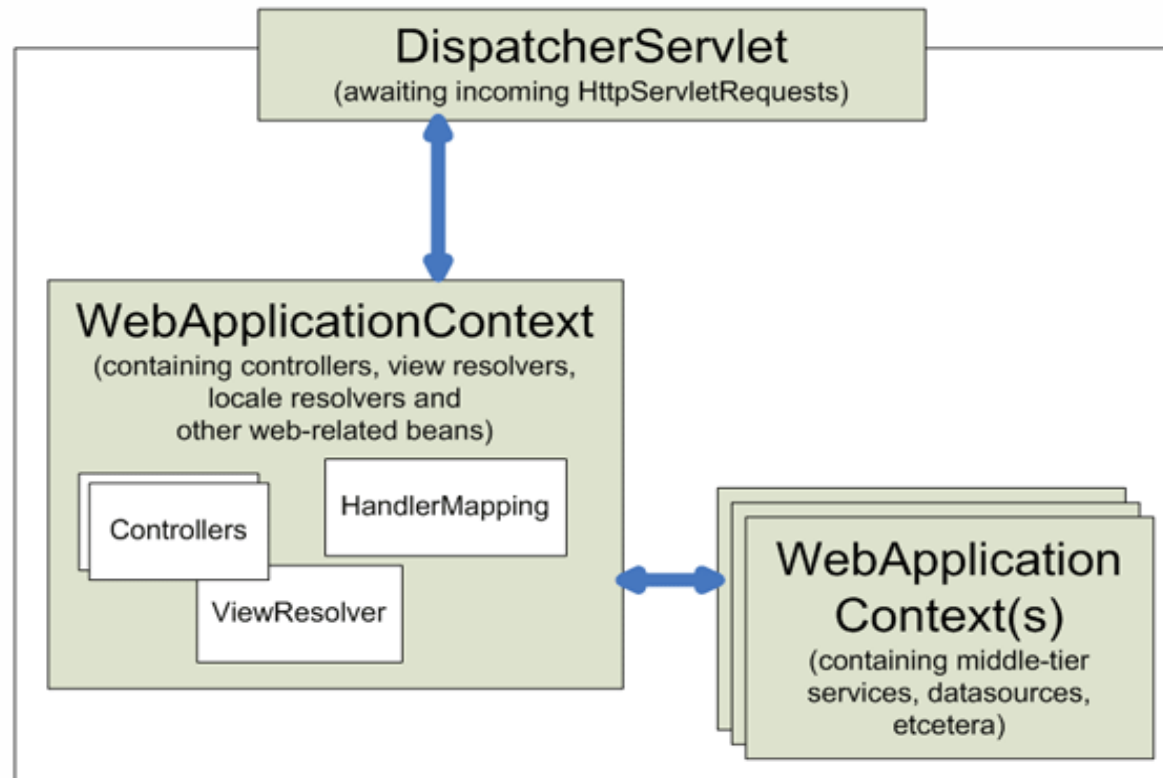
```
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/dispatcher-servlet-context.xml</param-value>
</context-param>

<servlet>
  <servlet-name>dispatcher-servlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value></param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher-servlet</servlet-name>
  <url-pattern>*</url-pattern>
</servlet-mapping>
```

- ❖ In the Web MVC framework, each **DispatcherServlet** has its own **WebApplicationContext**, which inherits all the beans already defined in the root **WebApplicationContext**.



# spring-servlet.xml configuration file

- ❖ WebApplicationContext will automatically look for the name as **spring-servlet.xml**.

```
<?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"
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">

<!--General config -->
<context:component-scan base-package="fa.training" />

<!-- Enable annotation -->
<context:annotation-config />

<!-- Enable web mvc -->
<mvc:annotation-driven />

<!-- Include JS or CSS files in a JSP page -->
<mvc:resources mapping="/resources/**" location="/resources/" />

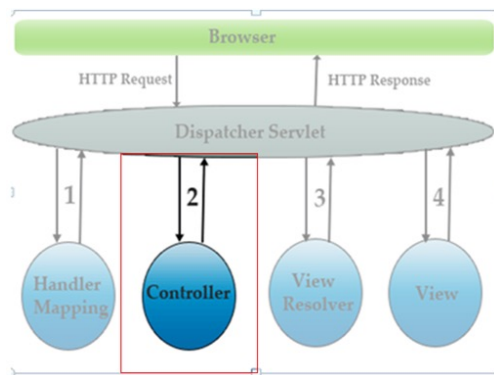
</beans>
```

## Section 2

# SPRING CONTROLLER

## ❖ Defining a controller with @Controller

- ✓ The **@Controller** annotation indicates that a particular class serves the role of a *controller*.
- ✓ The **@Controller** annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects **@RequestMapping** annotations.



```
HelloWorldController.java
1 package com.fsoft.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.servlet.ModelAndView;
6
7 @Controller
8 public class HelloWorldController {
9
10
11
12
13
14 }
```

## ❖ Mapping Requests With @RequestMapping

- ✓ Use the **@RequestMapping** annotation to map URLs such as /appointments onto an entire class or a particular handler method.

```
@RequestMapping( value = {"/hello"}, method= RequestMethod.GET)
public ModelAndView helloWorld() {
    String message = "HELLO SPRING MVC!";
    return new ModelAndView("hellopage", "message", message);
}
```



- ❖ `@GetMapping` is specialized version of `@RequestMapping` annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.GET)`.
- ❖ `@GetMapping` annotated methods handle the HTTP GET requests matched with given URI expression. e.g.

```
@GetMapping("/home")
public String homeInit(Model model) {
    return "home";
}

@GetMapping("/members/{id}")
public String getMembers(Model model) {
    return "member";
}
```

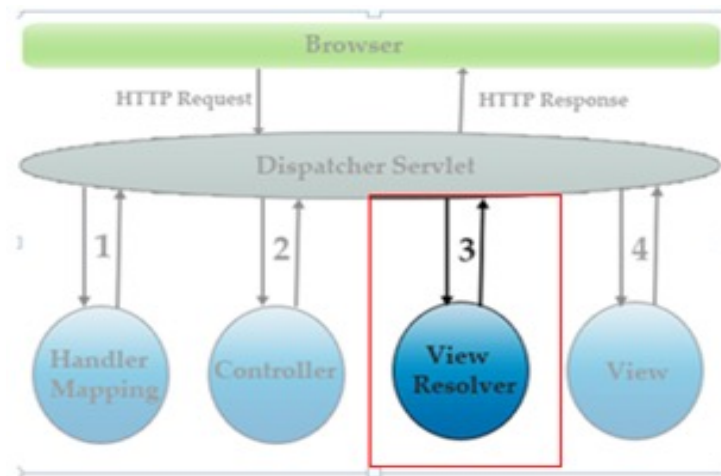
- ❖ @PostMapping is specialized version of @RequestMapping annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST).
- ❖ @PostMapping annotated methods handle the HTTP POST requests matched with given URI expression. e.g.

```
@PostMapping(path = "/members", consumes = "application/json", produces = "application/json")  
public void addMember(@RequestBody Member member) {  
    //code  
}
```

## Section 3

# RESOLVING VIEWS

- ❖ There are 2 interfaces that are important to the way Spring handle views: **ViewResolver** and **View**.
  - ✓ The **ViewResolver** provides a mapping between view names and actual views.
  - ✓ The **View** interface addresses the preparation of the request and hands the request over to one of the view technologies.



## ❖ Resolving views with the ViewResolver interface:

- ✓ Spring Web MVC controllers resolves logical view name either explicitly (e.g., by returning a **String**, **View**, or **ModelAndView**) or implicitly (i.e., based on conventions).
- ✓ In Spring, 'logical view name' represents Views and it is resolved by a view resolver.

## ❖ Resolving views with the ViewResolver interface:

ViewResolver	Description
<b>AbstractCachingViewResolver</b>	This view resolver caches views.
<b>XmlViewResolver</b>	This view resolver uses configuration file written in XML for view resolution.
<b>ResourceBundleViewResolver</b>	This view resolver use ResourceBundle, represented by bundle base name, to resolve view. Generally bundle is defined in a properties file, situated in the classpath.

ViewResolver	Description
<b>UrlBaseViewResolver</b>	This view resolver uses “logical view name” returned to find actual view.
<b>InternalResourceViewResolver</b>	This view resolver is the subclass of <code>UrlBasedViewResolver</code> and also support <code>InternalResourceView</code> and also subclass such as <code>JstlView</code> and <code>TilesView</code> .
<b>VelocityViewResolver/ FreeMarkerViewResolver</b>	This view resolver is the subclass of <code>UrlBasedViewResolver</code> which supports <code>VelocityView</code> , <code>FreeMarkerView</code> and its custom subclass.
<b>ContentNegotiatingViewResolver</b>	This view resolver is the implementation of the <code>ViewResolver</code> interface which resolves view on the basis of request file name or Accept header

- ❖ Create **spring-servlet.xml** under the **src/main/webapp/WEB-INF** folder and define the beans.
- ❖ **Example** : Configuration for **InternalResourceViewResolver**

```
<bean id="viewResolver"  
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
    <property name="prefix" value="/views/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```

- ❖ **Example** : Configuration for **TilesViewResolver**

```
<bean id="viewResolver"  
    class="org.springframework.web.servlet.view.tiles3.TilesViewResolver">  
    <property name="viewClass">  
    <value>  
        org.springframework.web.servlet.view.tiles3.TilesView  
    </value>  
    </property>  
</bean>
```

- ❖ **XmlViewResolver** is used to resolve “view name” based on view beans in the XML file.
- ❖ By default, XmlViewResolver will load the view beans from **/WEB-INF/views.xml**.
  - ✓ This location can be overridden through the “**location**” property,
- ❖ File **spring-servlet.xml**:

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver">  
    <property name="location">  
        <value>/WEB-INF/spring-views.xml</value>  
    </property>  
</bean>
```



- ❖ A controller class, returns a view, named “**WelcomePage**”.

```
public class WelcomeController extends AbstractController{

    @Override
    protected ModelAndView handleRequestInternal(
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        ModelAndView model = new ModelAndView("WelcomePage");

        return model;
    }
}
```

- ❖ The “**view bean**” is just a normal Spring bean declared in the Spring’s bean configuration file, where
  - ✓ “**id**” is the “view name” to resolve.
  - ✓ “**class**” is the type of the view.
  - ✓ “**url**” property is the view’s url location.
- ❖ *File : **spring-views.xml***

```
<bean id="WelcomePage"  
      class="org.springframework.web.servlet.view.JstlView">  
    <property name="url" value="/WEB-INF/pages/WelcomePage.jsp" />  
</bean>
```

# Resolving Views: ResourceBundleViewResolver

- ❖ **ResourceBundleViewResolver** is used to resolve “view named” based on view beans in “.properties” file.
- ❖ By default, ResourceBundleViewResolver will loads the view beans from file **views.properties**
  - ✓ This location can be overridden through the “**basename**” property.
- ❖ File ***spring-servlet.xml***:

```
<bean class="org.springframework.web.servlet.view.  
    ResourceBundleViewResolver">  
    <property name="basename" value="spring-views" />  
    <property name="order" value="0" />  
</bean>
```

- ❖ A controller class, returns a view, named “**WelcomePage**”.

```
public class WelcomeController extends AbstractController {  
  
    @Override  
    protected ModelAndView handleRequestInternal(  
        HttpServletRequest request, HttpServletResponse response)  
        throws Exception {  
  
        ModelAndView model = new ModelAndView("WelcomePage");  
  
        return model;  
    }  
}
```

# Resolving Views: ResourceBundleViewResolver

- ❖ Declare each view bean as a normal resource bundle style (key & message), where
  - ✓ “**WelcomePage**” is the view name to match.
  - ✓ “**.(class)**” is the type of view.
  - ✓ “**.url**” is the view’s URL location.
- ❖ File ***spring-views.properties***:

```
course_list.(class) = org.springframework.web.servlet.view.JstlView
course_list.url = /views/pages/course_list.jsp
WelcomePage.(class) = org.springframework.web.servlet.view.JstlView
WelcomePage.url = /views/views/WelcomePage.jsp
```

## Section 3

# SPRING @AUTOWIRED

# Enabling @Autowired Annotations

- ❖ The Spring framework **enables automatic dependency injection**.
- ❖ By declaring all the bean dependencies in a Spring configuration file, **Spring container can autowire relationships between collaborating beans**. This is called ***Spring bean autowiring***.
- ❖ To use Java-based configuration in our application, let's enable annotation-driven injection to load our Spring configuration:

```
@Configuration
@ComponentScan(basePackages = { "fa.training.controller" })
public class WebMvcConfig {}
```

//or to activate the dependency injection annotations in Spring XML files.

```
<context:component-scan base-package=" fa.training.controller"/>
```

# @Autowired on Properties

- ❖ Let's see how we can annotate a property using *@Autowired*. This eliminates the need for getters and setters.
- ❖ First, let's define a *fooFormatter* bean:

```
@Component("fooFormatter")
public class FooFormatter {
    public String format() {
        return "foo";
    }
}
```

- ❖ Then, we'll inject this bean into the *FooService* bean using *@Autowired* on the field definition:

```
@Component
public class FooService {
    @Autowired
    private FooFormatter fooFormatter;
}
```



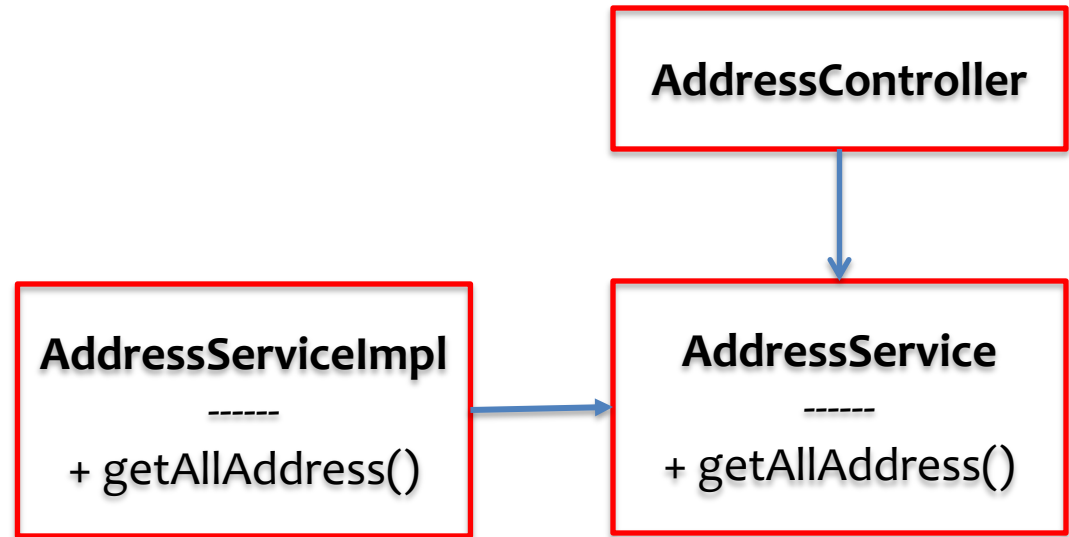
- ❖ The setter method is called with the instance of *FooFormatter* when *FooService* is created:

```
public class FooService {  
    private FooFormatter fooFormatter;  
    @Autowired  
    public void setFooFormatter(FooFormatter fooFormatter) {  
        this.fooFormatter = fooFormatter;  
    }  
}
```

- ❖ An instance of *FooFormatter* is injected by Spring as an argument to the *FooService* constructor:

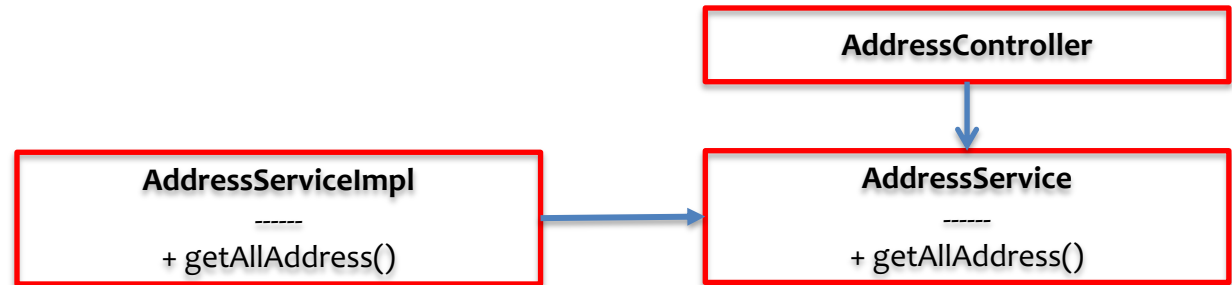
```
public class FooService {  
    private FooFormatter fooFormatter;  
    @Autowired  
    public FooService(FooFormatter fooFormatter) {  
        this.fooFormatter = fooFormatter;  
    }  
}
```

## ❖ With Standard code:



```
public class AddressController {  
    AddressService addressService = new AddressServiceImpl();  
  
    public List<Address> getAllAddress(){  
        return addressService.getAllAddress();  
    }  
}
```

## ❖ With Autowire:



```
@Controller
public class AddressController {

    @Autowired
    private AddressService addressService;

    @RequestMapping(value="/addressList", method= RequestMethod.GET)
    public String getAllAddress(ModelMap modelMap){
        List<Address> listOfAddress = addressService.getAllAddress();
        modelMap.addAttribute("addresses", listOfAddress);

        return "address_list";
    }
}
```

- ❖ By default, Spring resolves `@Autowired` entries by type. **If more than one bean of the same type is available in the container, the framework will throw a fatal exception.**
- ❖ To resolve this conflict, we need to tell Spring explicitly which bean we want to inject.
- ❖ **Autowiring by `@Qualifier`**

```
@Component("fooFormatter")
public class FooFormatter implements Formatter {
    public String format() {
        return "foo";
    }
}

@Component("barFormatter")
public class BarFormatter implements Formatter {
    public String format() {
        return "bar";
    }
}
```

```
public class FooService {
    @Autowired
    @Qualifier("fooFormatter")
    private Formatter formatter;
}
```

## ❖ Autowiring by Name

```
@Component("fooFormatter")
public class FooFormatter implements Formatter {
    public String format() {
        return "foo";
    }
}
```

```
@Component("barFormatter")
public class BarFormatter implements Formatter {
    public String format() {
        return "bar";
    }
}
```

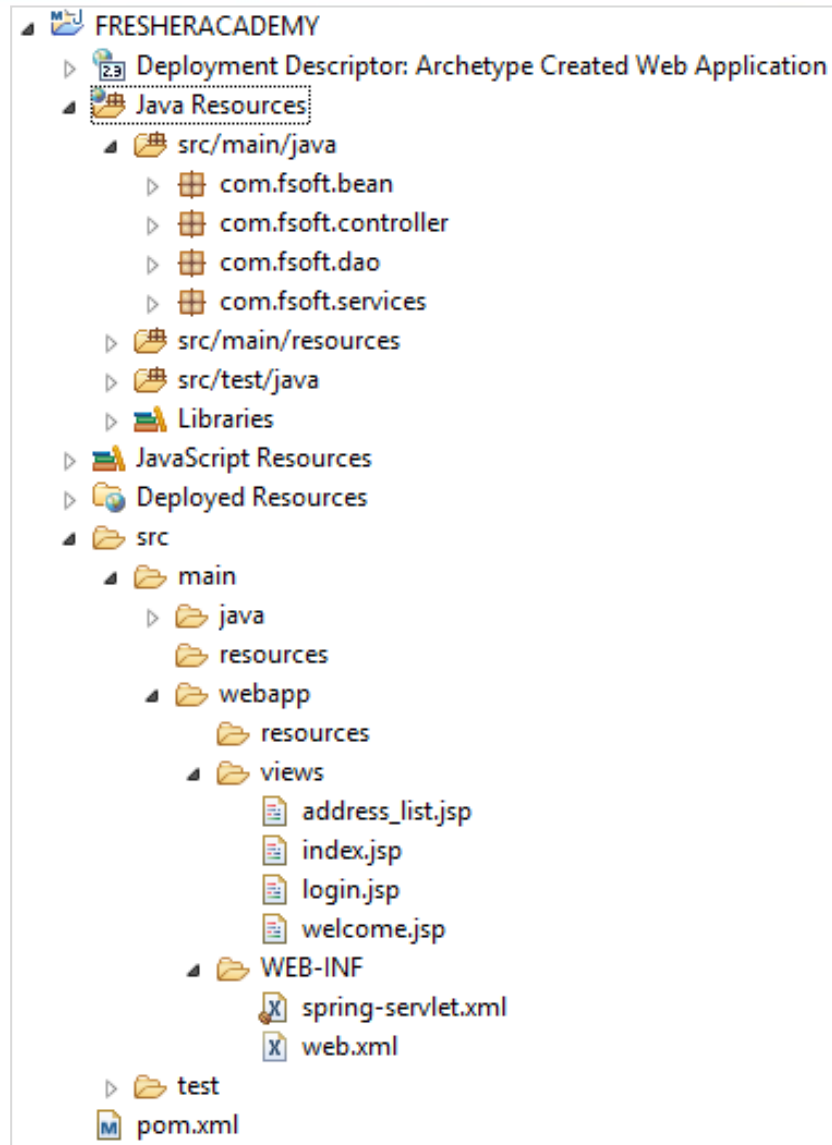
```
public class FooService {
    @Autowired
    private Formatter
    fooFormatter;
}
```

## Section 4

# SPRING MVC FIRST EXAMPLE

- ❖ There are given 5 steps for creating the spring MVC application.
- ❖ The steps are as follows:
  - ✓ **Create the request /response pages**
  - ✓ **Create the bean/controller/service/dao class**
  - ✓ **Provide the entry of controller in the web.xml file**
  - ✓ **Define the bean in the xml file**
  - ✓ **Start server and deploy the project**

# Directory Structure using Maven





## ❖ Add maven dependency in pom.xml file.

```
<properties>
    <spring.version>5.2.10.RELEASE</spring.version>
</properties>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- Spring MVC Dependency -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
</dependency>
```

Áp dụng cho **XML Configuration**

## ❖ Add maven dependency in pom.xml file.

```
<!-- Spring JDBC -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- JSTL -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<!-- MS SQL Server -->
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
    <version>7.4.1.jre8</version>
</dependency>
</dependencies>
<build>
    <finalName>FRESHERACADEMY</finalName>
</build>
</project>
```

▲ You can add the needful dependencies.

- ❖ We are assuming that you have created the following table inside the **FAMS** database.

```
CREATE TABLE [dbo].[Users](  
    [user_id] [int] IDENTITY(1,1) PRIMARY KEY NOT NULL,  
    [user_name] [varchar](50) NOT NULL,  
    [email] [varchar](50) NOT NULL,  
    [password] [varchar](50) NOT NULL,  
    [enabled] TINYINT NOT NULL DEFAULT 1  
)
```

- ❖ In Spring MVC, The *DispatcherServlet* needs to be declared and mapped for processing all requests either using java or **web.xml** configuration.
- ❖ In a Servlet 3.0+ environment, you can use ***AbstractAnnotationConfigDispatcherServletInitializer*** class to register and initialize the DispatcherServlet programmatically as follows.

```
public class AppInitializer
    extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] {};
        // return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { WebMvcConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

# (1) Create the request/response pages

## ❖ Create **login.jsp**:

```
<body>
<form action="Login" method="post" name="LoginForm">
    User Name: <input type="text" name="userName">
    Password:  <input type="password" name="password">
               <input type="submit" value="Login">
</form>
</body>
```

## ❖ And **index.jsp**:

```
<body>
    <h2>Welcome <span>${userName}</span>! </h2>
</body>
</html>
```

## (2) Create the bean class

```
UserController.java  User.java ✕
1 package com.fsoft.bean;
2
3 public class User {
4     private String userName, password;
5
6     public User() {}
7
8
9
10    public User(String userName, String password) {
11        this.userName = userName;
12        this.password = password;
13    }
14
15    public String getUserName() {
16        return userName;
17    }
18
19    public void setUserName(String userName) {
20        this.userName = userName;
21    }
22
23    public String getPassword() {
24        return password;
25    }
26
27    public void setPassword(String password) {
28        this.password = password;
29    }
30 }
31
```

## (2) Create the controller class

```
UserController.java
1 package com.fsoft.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.web.bind.annotation.ModelAttribute;
6 import org.springframework.web.bind.annotation.RequestMapping;
7 import org.springframework.web.bind.annotation.RequestMethod;
8 import org.springframework.web.servlet.ModelAndView;
9
10 import com.fsoft.bean.User;
11 import com.fsoft.services.UserService;
12
13 @Controller
14 public class UserController {
15     @Autowired
16     private UserService userService;
17
18     /**
19      *
20      * @param user
21      * @param modelMap
22      * @return String value
23      */
24
25     @RequestMapping(value = { "/login" }, method = RequestMethod.POST)
26     public ModelAndView chekLogin(@ModelAttribute("login") User user) {
27         User userData = userService.checkLogin(user);
28         if (userData != null) {
29
30             return new ModelAndView("index", "userName", userData.getUserName());
31         } else {
32             return new ModelAndView("login", "message", "Login Fail!");
33         }
34     }
35 }
36
```

## (2) Create the service class

```
UserController.java  UserService.java  UserServiceImpl.java
1 package com.fsoft.services;
2
3 import com.fsoft.bean.User;
4
5 public interface UserService {
6     public User checkLogin(User user);
7 }
8
```

```
UserController.java  User.java  UserService.java  UserServiceImpl.java
1 package com.fsoft.services;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5 import com.fsoft.bean.User;
6 import com.fsoft.dao.UserDao;
7
8 public class UserServiceImpl implements UserService {
9     @Autowired
10     private UserDao userDao;
11
12     public User checkLogin(User user) {
13         return userDao.checkLogin(user);
14     }
15 }
16
```



## (2) Create the dao class

```
UserController.java  User.java  UserDao.java X
1 package com.fsoft.dao;
2
3 import com.fsoft.bean.User;
4
5 public interface UserDao {
6     public User checkLogin(User user);
7 }
8
```

```
UserController.java  UserDaoImpl.java X
1 package com.fsoft.dao;
2
3 import java.sql.ResultSet;
4
5 public class UserDaoImpl implements UserDao {
6
7     @Autowired
8     private JdbcTemplate jdbcTemplate;
9
10    public User checkLogin(User user) {
11        String query = "SELECT * FROM dbo.Users WHERE user_name = ? AND password = ?";
12        List<User> listUser = jdbcTemplate.query(query,
13            new Object[] { user.getUserName(), user.getPassword() }, new UserRowMapper());
14
15        return listUser.get(0);
16    }
17 }
18
19 class UserRowMapper implements RowMapper<User>{
20
21     public User mapRow(ResultSet resultSet, int iValue) throws SQLException {
22         User user = new User();
23         user.setUserName(resultSet.getString(2));
24         user.setPassword(resultSet.getString(3));
25         return user;
26     }
27 }
28
```

### (3) Provide the entry of controller in the web.xml

```
UserController.java  web.xml
1 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xmlns="http://java.sun.com/xml/ns/javaee"
3   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
4   version="3.0">
5   <display-name>Archetype Created Web Application</display-name>
6   <welcome-file-list>
7     <welcome-file>/views/login.jsp</welcome-file>
8   </welcome-file-list>
9
10  <servlet>
11    <servlet-name>spring</servlet-name>
12    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
13    <init-param>
14      <param-name>contextConfigLocation</param-name>
15      <param-value>/WEB-INF/spring-servlet.xml</param-value>
16    </init-param>
17    <load-on-startup>1</load-on-startup>
18  </servlet>
19
20  <servlet-mapping>
21    <servlet-name>spring</servlet-name>
22    <url-pattern>/</url-pattern>
23  </servlet-mapping>
24 </web-app>
```

## (4) Define the bean in the xml file

- ❖ Create **spring-servlet.xml** under the **/WEB-INF** folder and define the beans.

```
UserController.java  web.xml  spring-servlet.xml  ✕
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4      xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xmlns:jdbc="http://www.springframework.org/schema/jdbc" xmlns:tx="http://www.springframework.org/schema/tx"
6      xmlns:context="http://www.springframework.org/schema/context"
7      xsi:schemaLocation="http://www.springframework.org/schema/jdbc
8          http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
9          http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd
11         http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
12         http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
13         http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
14
15
16     <!-- <import resource="classpath:user-beans.xml" /> -->
17     <context:component-scan base-package="com.fsoft" />
18     <context:annotation-config />
19     <mvc:annotation-driven />
20     <!-- Bean -->
21
22     <bean id="userDao" class="com.fsoft.dao.UserDaoImpl" />
23     <bean id="userService" class="com.fsoft.services.UserServiceImpl" />
24
25     <!-- Datasource -->
26     <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
27         <property name="dataSource" ref="dataSource" />
28     </bean>
```

## (4) Define the bean in the xml file

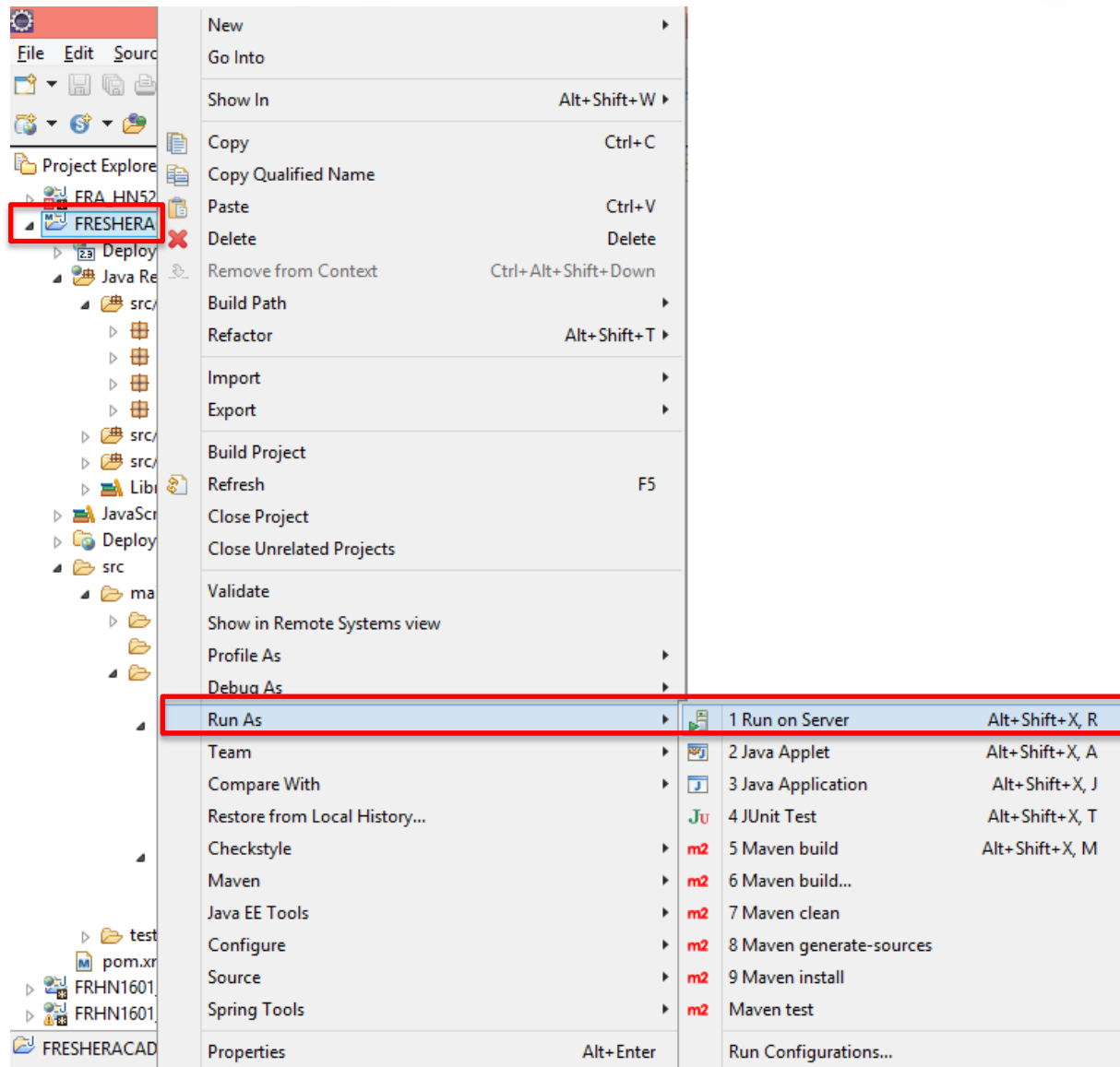
```
29
30 <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
31     <property name="driverClassName" value="com.microsoft.sqlserver.jdbc.SQLServerDriver" />
32     <property name="url" value="jdbc:sqlserver://1WDDIEUNT1-LT:1433;databaseName=FAMS" />
33     <property name="username" value="sa" />
34     <property name="password" value="12345678" />
35 </bean>
36 <!-- View Resolver-->
37 <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
38     <property name="prefix" value="/views/" />
39     <property name="suffix" value=".jsp" />
40 </bean>
41
42 <mvc:resources mapping="/resources/**" location="/resources/" /><!--cache-period="31556926 -->
43 </beans>
44 <
```

```
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = { "fa.training.controller" })
public class WebMvcConfig implements WebMvcConfigurer {
    @Bean
    public InternalResourceViewResolver resolver() {
        InternalResourceViewResolver resolver =
            new InternalResourceViewResolver();
        resolver.setViewClass(JstlView.class);
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```

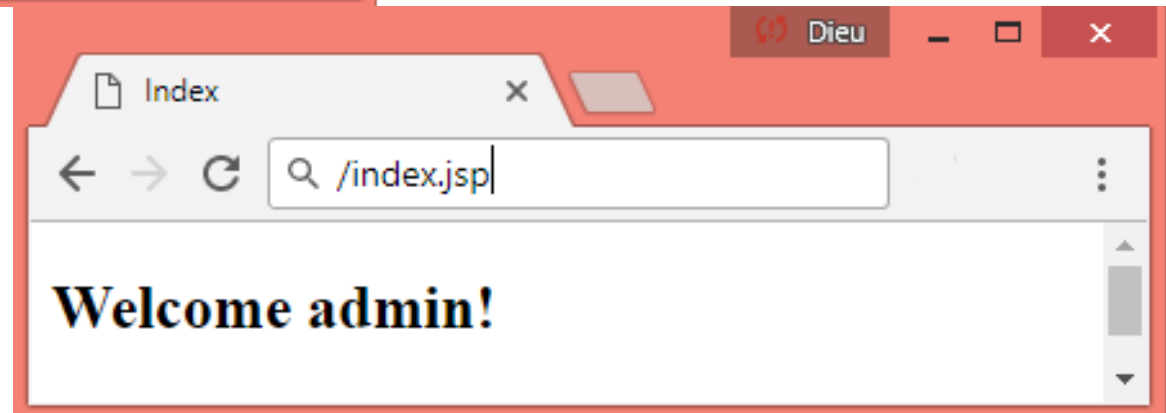
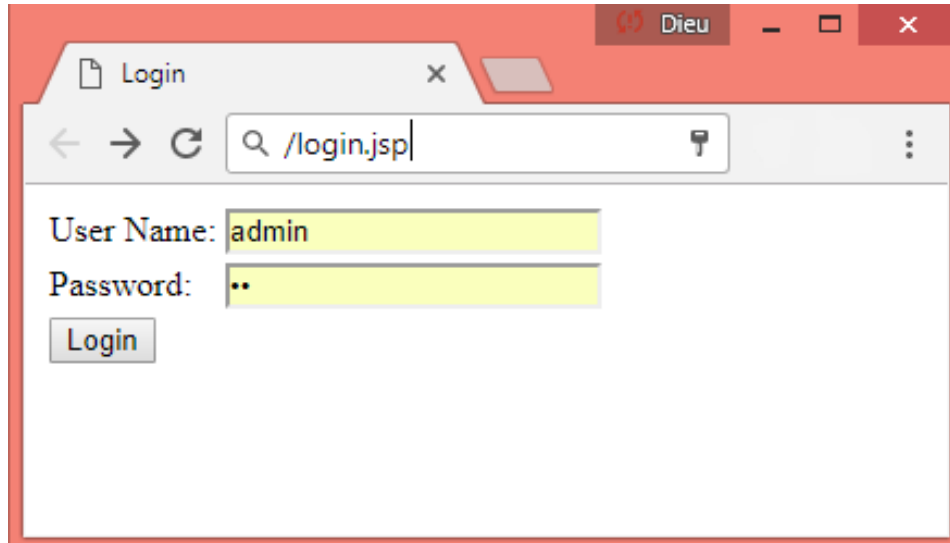
## ❖ Annotation configuration:

- ✓ `@EnableWebMvc`: enables default Spring MVC configuration and registers Spring MVC infrastructure components expected by the `DispatcherServlet`.
- ✓ `@Configuration`: indicates that a class declares one or more `@Bean` methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.
- ✓ `@ComponentScan` annotation is used to specify the base packages to scan. Any class which is annotated with `@Component` and `@Configuration` will be scanned.
- ✓ `InternalResourceViewResolver` helps in mapping the logical view names to directly view files under a certain pre-configured directory.

# (5) Start server and deploy the project



## (5) Start server and deploy the project





1

- **Introduction to Spring Web MVC**

2

- **Spring Controller**

3

- **Resolving Views**

4

- **Spring @Autowired**

5

- **Spring MVC First Example**

# Thank you

