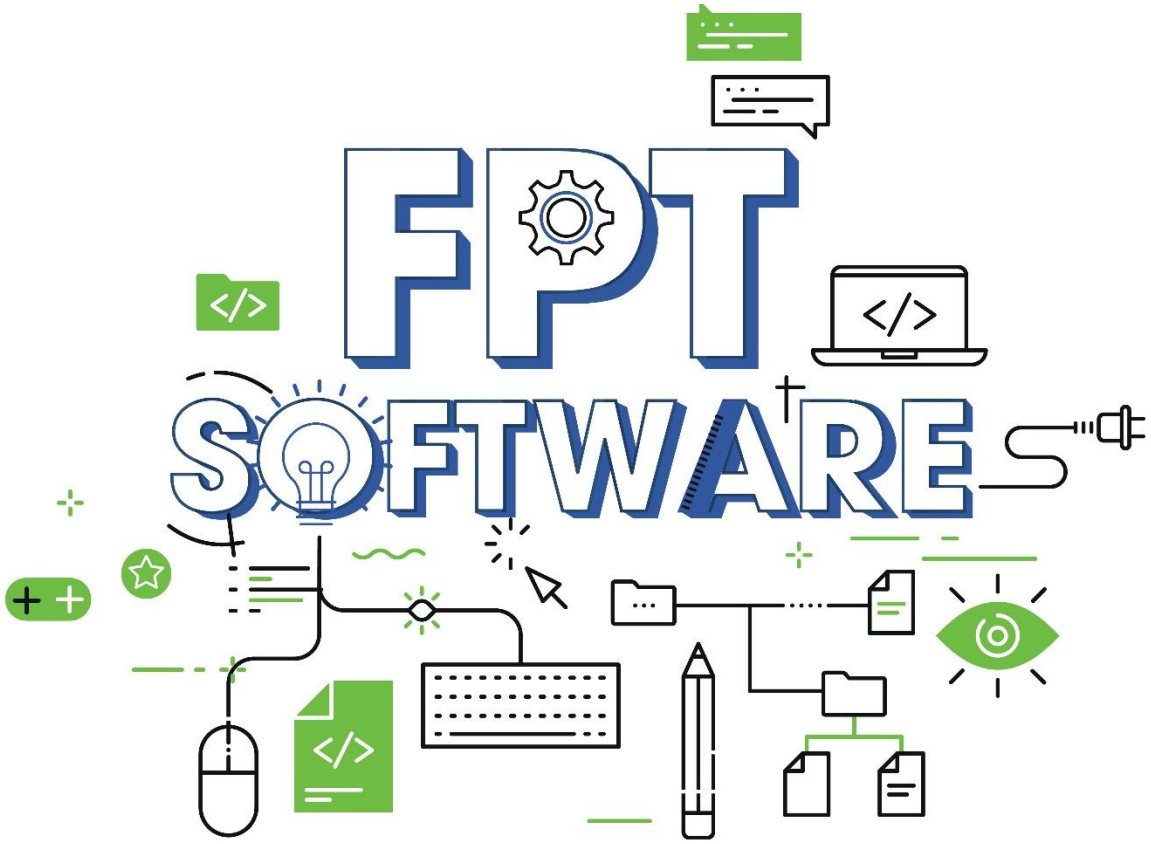
# SPRING WEBMVC INTRODUCTION




Design by: DieuNT1

# Agenda

1. Introduction to Spring Web MVC
2. Spring Controller
3. Resolving Views
4. Spring @Autowired
5. Spring MVC First Example

# Learning Goals

- **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

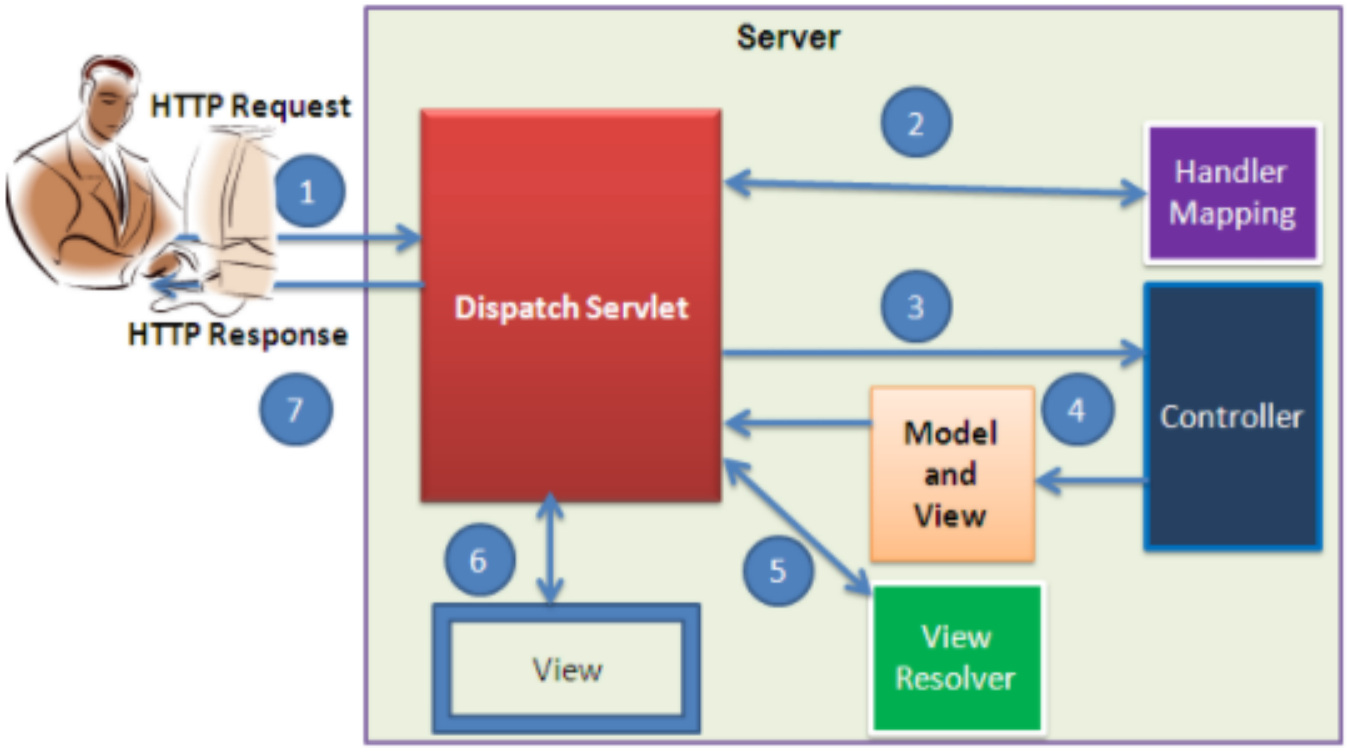
# SPRING WEB MVC FRAMEWORK

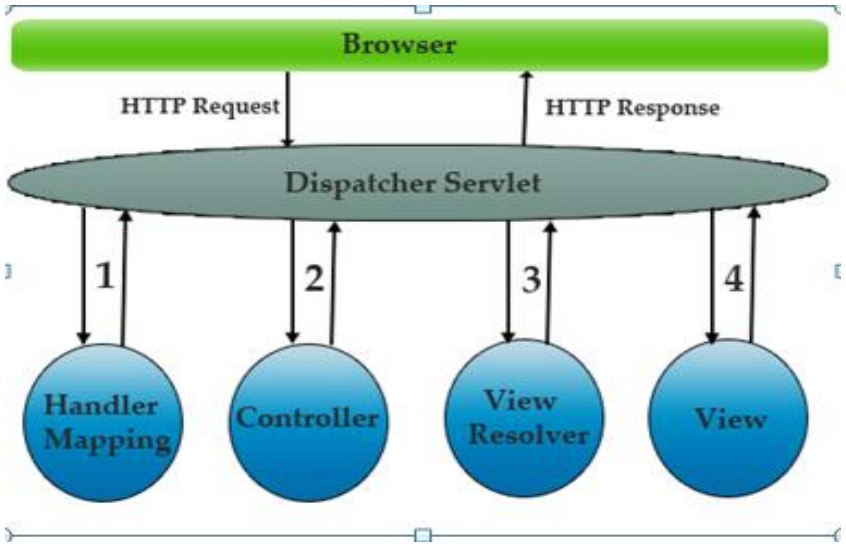
# Introduction to Spring MVC


- A part of the **Spring Framework** is **Spring Web MVC**, an extensible MVC framework for creating web applications.
- Support for **I**nvesrsion **o**f **C**ontrol (or, the Dependency Injection).
- Use **DispatcherServlet** that dispatches request to handler.
- The default handler is based on the **@Controller** and **@RequestMapping** annotations.

# An Introduction to MVC

- **A Web Application developed using SpringMVC framework:**



# 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** objectthat 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.

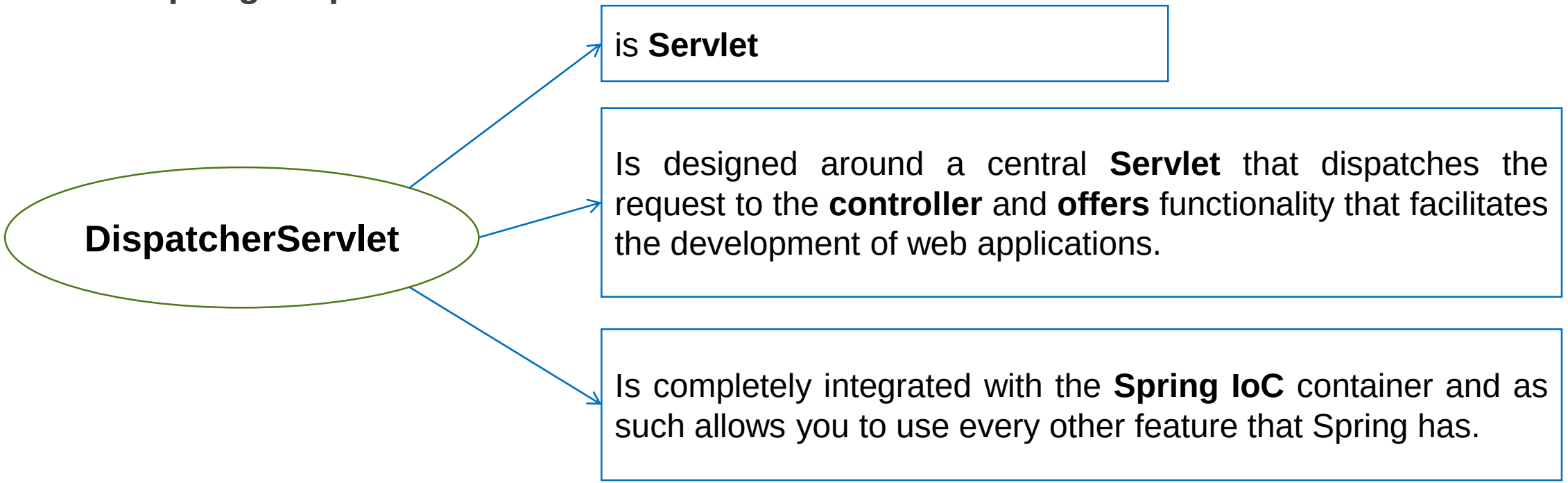
**Step5**
- The DispatcherServlet will send the view name to a **ViewResolver** to find the original View (.jsp) page to display.

**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.

# DispatcherServlet

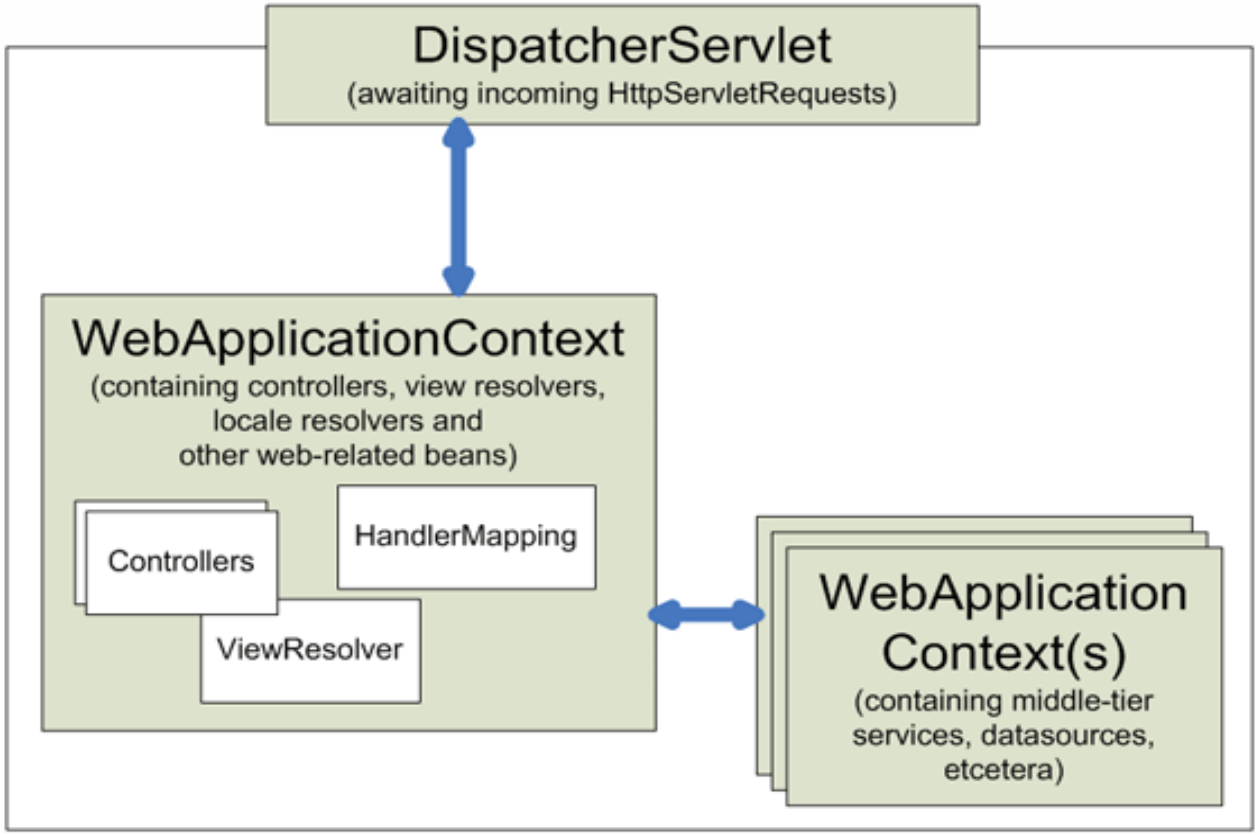
- **What is Spring DispatcherServlet?**




- ✓ **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.

# DispatcherServlet and WebApplicationContext

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



Section 1.1

# DispatcherServlet: XML Configuration

# DispatcherServlet

- 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>
```

# WebApplicationContext

- **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>
```

# dispatcher-servlet.xml configuration file

- **WebApplicationContext** will automatically look for the name as **dispatcher-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 1.2

# DispatcherServlet: Java Configuration

# Register a DispatcherServlet

- **AbstractAnnotationConfigDispatcherServletInitializer** class implements **WebMvcConfigurer** which internally implements WebApplicationInitializer.
- It registers a **ContextLoaderListener** (optionally) and a **DispatcherServlet** and allows you to easily **add configuration classes to load for both classes and to apply filters to the DispatcherServlet** and provide the servlet mapping.

```
public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

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

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

# Configuring Spring MVC

- Spring MVC configuration using annotations is as follows.

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "fa.training" })
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(final ResourceHandlerRegistry registry) {

        registry.addResourceHandler("/resources/**").addResourceLocations("/resources/")
                                                    .setCachePeriod(0);

    }
}
```

# Configuring Spring MVC

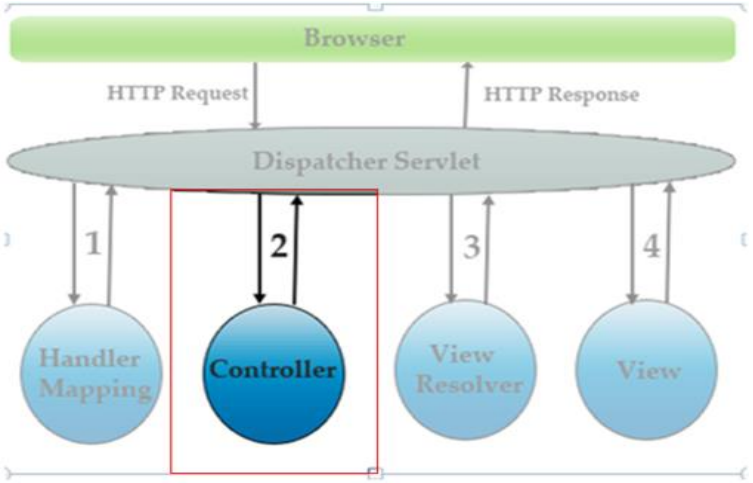
- **WebMvcConfigurer** defines options for customizing or adding to the default Spring MVC configuration enabled through the use of @EnableWebMvc.
- **@EnableWebMvc** enables default Spring MVC configuration and registers Spring MVC infrastructure components expected by the **DispatcherServlet**.
- **@Configuration** 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.
- **ResourceBundleMessageSource** accesses resource bundles using specified basenames (here it is messages).

Section 2

# Spring Controller

# Implementing Controllers

- **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 }
```

# Implementing Controllers

- **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);
}
```

# Spring Controller

- `@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.

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

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

# Spring Controller

- `@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.

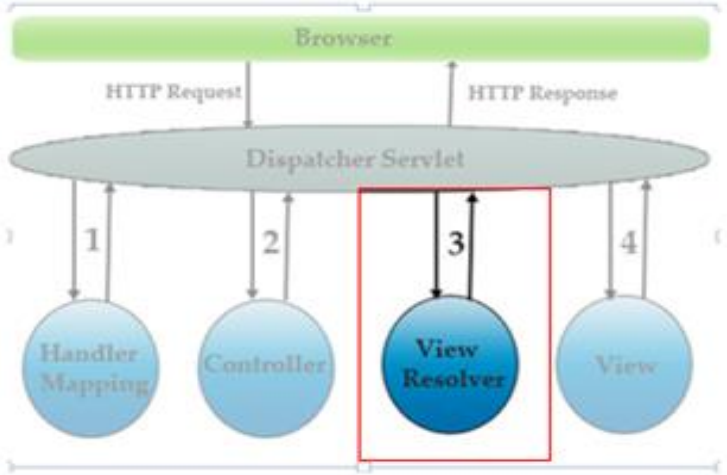
```
@PostMapping(path = "/members", consumes = "application/json", produces = "application/json")
public void addMember(@RequestBody Member member) {
    //code
}
```

Section 3

# Resolving Views

# 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

- **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 |
|---|---|
| **AbstractCachingViewResolver** | This view resolver caches views. |
| **XmlViewResolver** | This view resolver uses configuration file written in XML for view resolution. |
| **ResourceBundleViewResolver** | 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. |

# Resolving Views

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

# Resolving Views: XML Configuration

- In **dispatcher-servlet.xml** file:
- **Example** : Configuration for **InternalResourceViewResolver**

```
<bean id="viewResolver"
                class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/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>
```

# Resolving Views: XML Configuration

- **XmlViewResolver** is used to resolve “view name” based on view beans in the XML file.
- By default, XmlViewResolver will loads 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>
```

# Resolving Views: XmlViewResolver

- 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: Java Configuration

- **In WebConfig class:**

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "fa.training"})
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;
    }
}
```

# Resolving Views: Controller

- 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"); // /WEB-INF/views/WelcomePage.jsp

    return model;
  }
}
```

Section 3

# Spring @Autowired

# Enabling @*Autowired* Annotations

- The Spring framework enables automatic dependency injection.
- We learned *@Autowired* annotation to inject the dependency automatically using **Constructor injection**, **Setter injection**, and **Field injection**.
- To use Java-based configuration in our application, let's enable annotation-driven injection to load our Spring configuration:

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

//or to activate the dependency injection annotations in Spring XML files.
<context:annotation-config />
<context:component-scan base-package=" fa.training"/>
```

# Field Injection using @Autowired Annotation

- **Example**:

```
public interface Pizza {
        String getPizza();
}
```

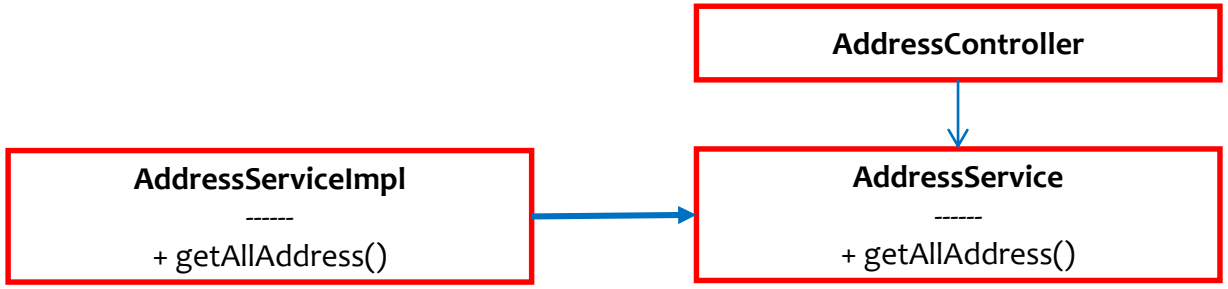
```
@Component public class VegPizza implements Pizza {
        @Override
        public String getPizza() {
                return "Veg Pizza";
        }
}
```

```
@Component
public class NonVegPizza implements Pizza {
        @Override
        public String getPizza() {
                return "Non-veg Pizza";
        }
}
```

```
@Component
public class PizzaController {

        @Autowired
        @Qualifier("vegPizza")
        private Pizza pizza;

        public String getPizza(){
                return pizza.getPizza();
        }
}
```

# Field Injection

- 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";
}
}
```

# Setters and Constructors Injection

- **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;
    }
}
```

# Autowire Disambiguation

- 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;
}
```

# Autowire Disambiguation

- **Autowiring byName and constructor:**

```
@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;
}
```

```
public class FooService {
          private Formatter fooFormatter;

          public Formatter(Formatter fooFormatter) {
                this. fooFormatter = fooFormatter;
          }
}
```

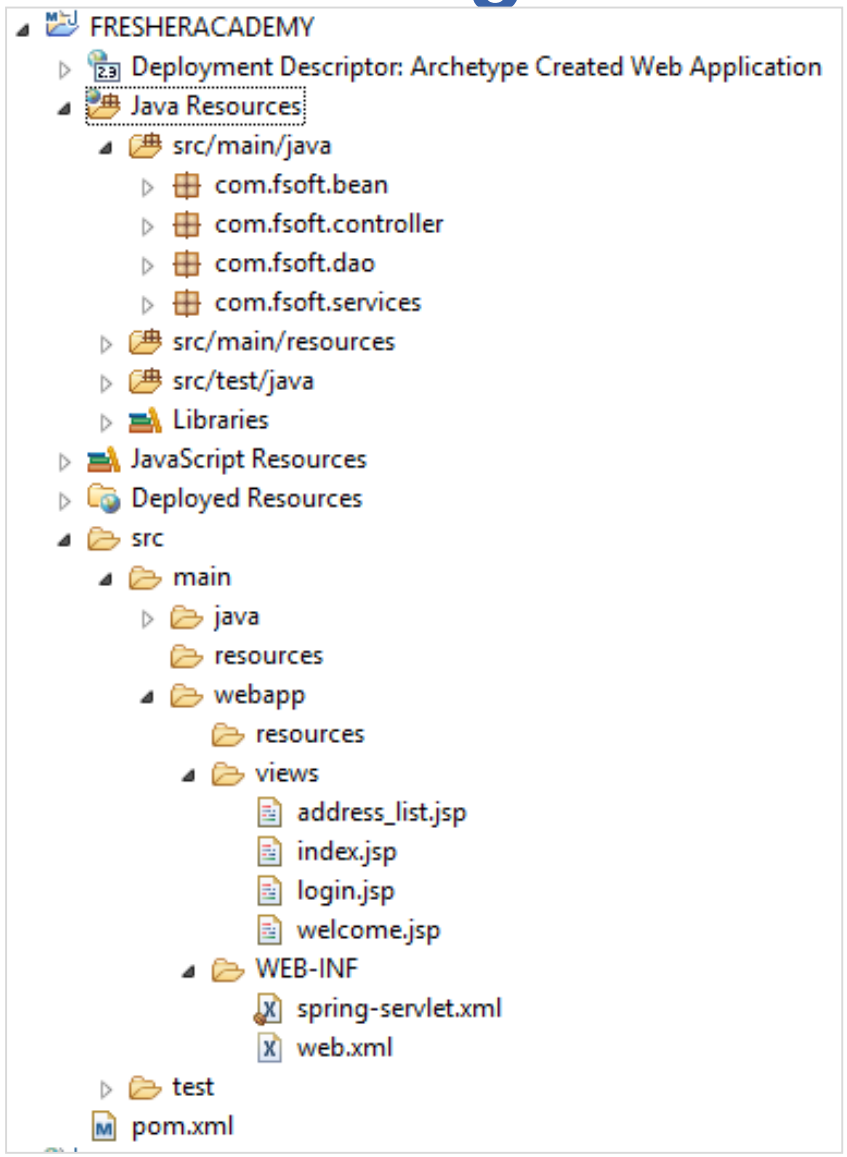
Section 4

# Spring MVC First Example

# Spring MVC First Example

- There are given 5 steps for creating the spring MVC application.
- The steps are as follows:
  - ✓ **Create the request /respone 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



# Required Maven Dependency

- **Add maven dependency in pom.xml file.**

```
<properties>
          <spring.version>6.0.10.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**

# Required Maven Dependency

- **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>11.2.3.jre8</version>
</dependency>
</dependencies>

<build>
          <finalName>FRESHERACADEMY</finalName>
</build>
</project>
```

- *You can add the needful dependencies.*

# Table definition

- 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
)
```

# (1) Create the request/respone 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

```java
package com.fsoft.bean;

public class User {
    private String userName, password;

    public User() {

    }

    public User(String userName, String password) {
        this.userName = userName;
        this.password = password;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

# (2) Create the controller class

UserController.java

```java
package com.fsoft.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import com.fsoft.bean.User;
import com.fsoft.services.UserService;

@Controller
public class UserController {
    @Autowired
    private UserService userService;

    /**
     *
     * @param user
     * @param modelMap
     * @return String value
     *
     */
    @RequestMapping(value = { "/login" }, method = RequestMethod.POST)
    public ModelAndView chekLogin(@ModelAttribute("login") User user) {
        User userData = userService.checkLogin(user);
        if (userData != null) {

            return new ModelAndView("index", "userName", userData.getUserName());
        } else {
            return new ModelAndView("login", "message", "Login Fail!");
        }
    }
}
```

# (2) Create the service class

UserController.java | UserService.java | UserServiceImpl.java

```java
package com.fsoft.services;

import com.fsoft.bean.User;

public interface UserService {
    public User checkLogin(User user);
}

```

UserController.java | User.java | UserService.java | UserServiceImpl.java

```java
package com.fsoft.services;

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

import com.fsoft.bean.User;
import com.fsoft.dao.UserDao;

public class UserServiceImpl implements UserService {
    @Autowired
    private UserDao userDao;

    public User checkLogin(User user) {
        return userDao.checkLogin(user);
    }
}

```

# (2) Create the dao class

UserController.java | User.java | UserDao.java

```java
package com.fsoft.dao;

import com.fsoft.bean.User;

public interface UserDao {
    public User checkLogin(User user);
}

```

UserController.java | UserDaoImpl.java

```java
package com.fsoft.dao;

import java.sql.ResultSet;

public class UserDaoImpl implements UserDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;


    public User checkLogin(User user) {
        String query = "SELECT * FROM dbo.Users WHERE user_name = ? AND password = ?";
        List<User> listUser = jdbcTemplate.query(query,
                new Object[] { user.getUserName(), user.getPassword() }, new UserRowMapper());

        return listUser.get(0);
    }

}

class UserRowMapper implements RowMapper<User>{

    public User mapRow(ResultSet resultSet, int iValue) throws SQLException {
        User user = new User();
        user.setUserName(resultSet.getString(2));
        user.setPassword(resultSet.getString(3));
        return user;
    }

}
```

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

UserController.java | 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>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

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

# (4) Define the bean in the xml file

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

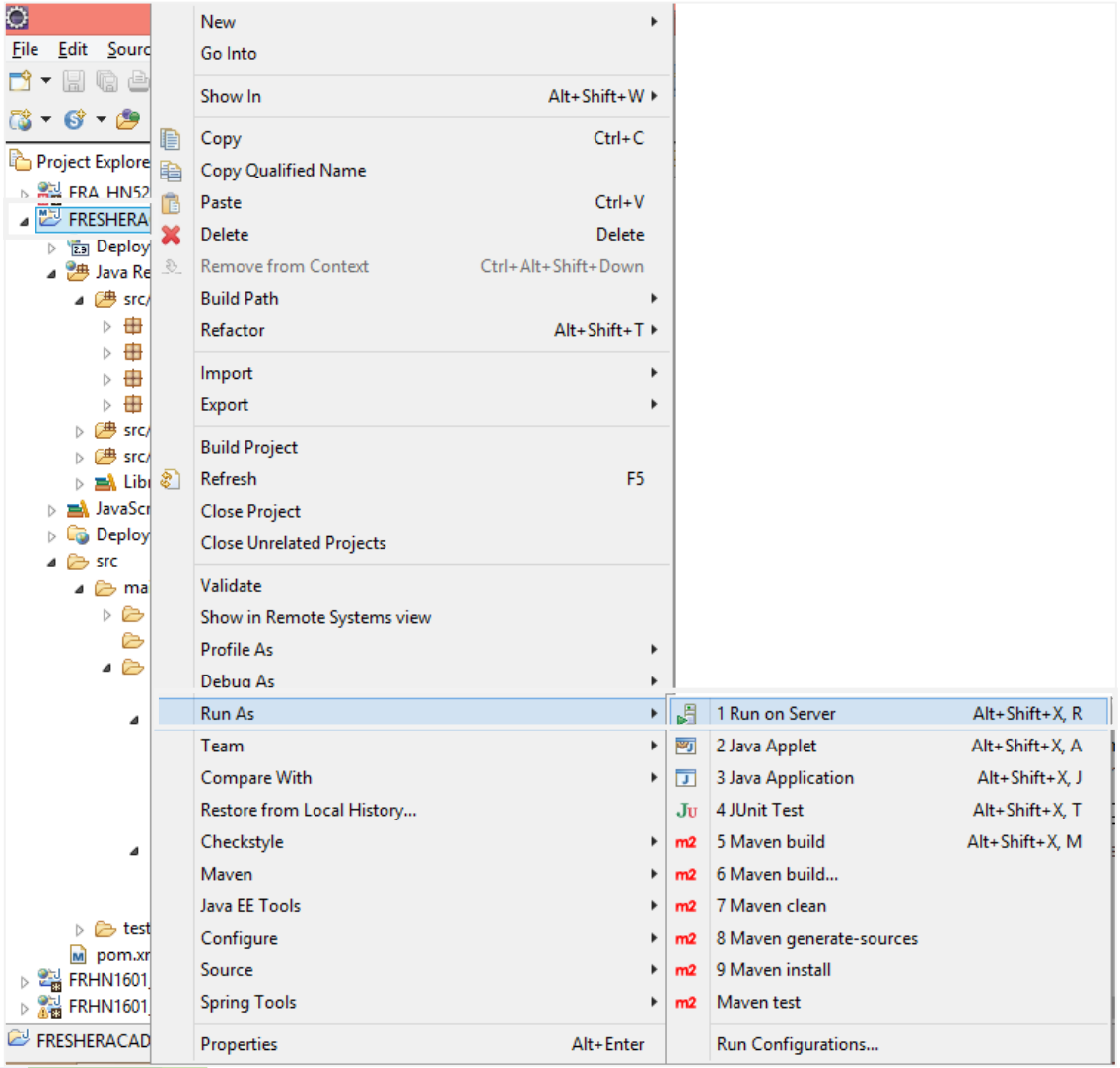
UserController.java | web.xml | spring-servlet.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc" xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.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-4.0.xsd
  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">


    <!-- <import resource="classpath:user-beans.xml" /> -->
    <context:component-scan base-package="com.fsoft" />
    <context:annotation-config />
    <mvc:annotation-driven />
    <!-- Bean -->

    <bean id="userDao" class="com.fsoft.dao.UserDaoImpl" />
    <bean id="userService" class="com.fsoft.services.UserServiceImpl" />

    <!-- Datasourse -->

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

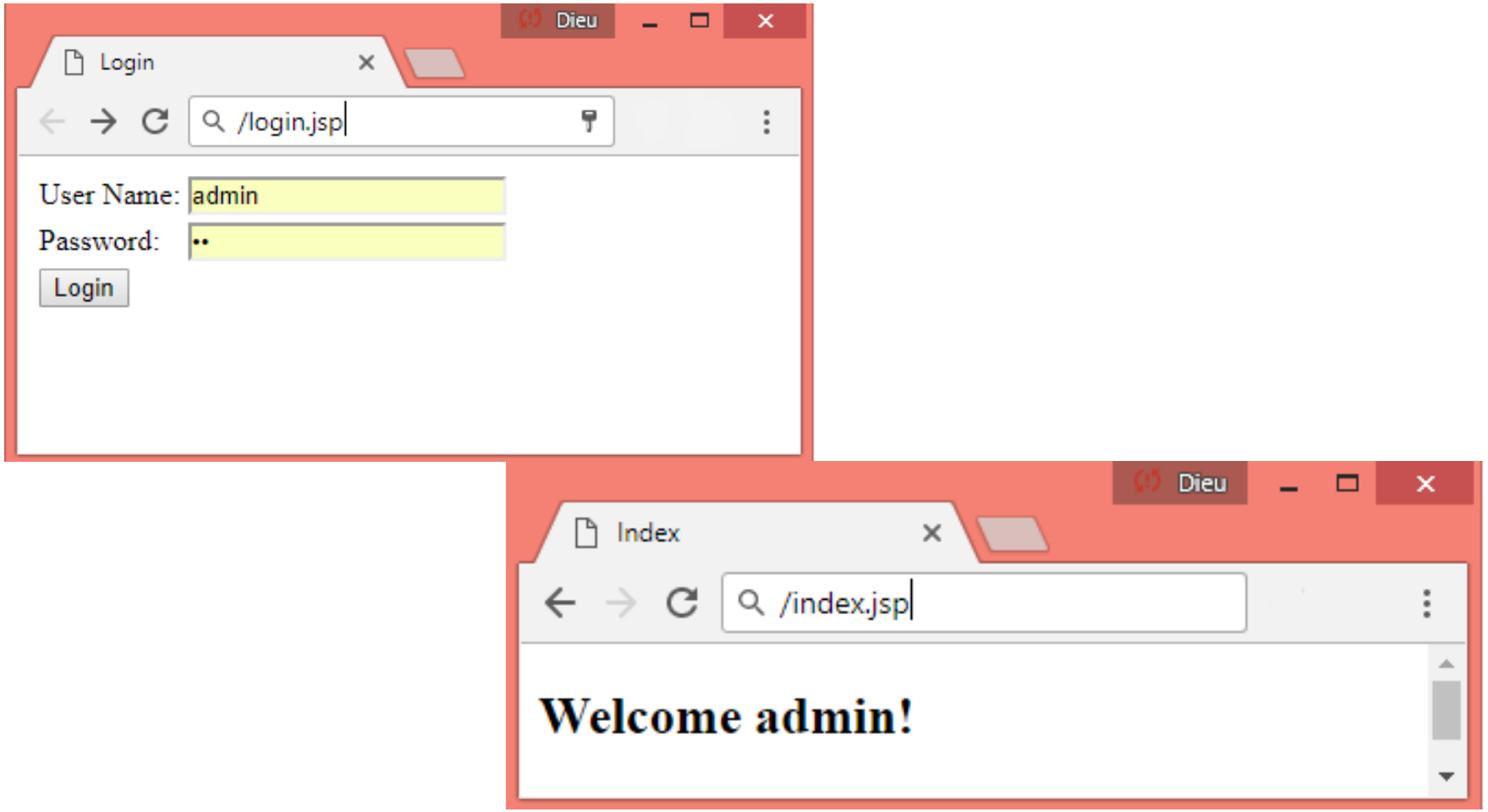
# (4) Define the bean in the xml file

```
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>
```

# (5) Start server and deploy the project



# (5) Start server and deploy the project



# Summary

- Introduction to Spring Web MVC
- Spring Controller
- Resolving Views
- Spring @Autowired
- Spring MVC First Example

# THANK YOU!