

Spring MVC

Objectives

- ◆ Spring MVC Basics
- ◆ Spring MVC Framework
- ◆ Controller/Model/View
- ◆ Spring Interceptor
- ◆ Spring Validator

The MVC pattern

- ◆ MVC pattern breaks an application into three parts:
 - Model: The domain object model / service layer
 - View: Template code / markup
 - Controller: Presentation logic / action classes
- ◆ MVC defines interaction between components to promote separation of concerns and loose coupling
 - Each file has one responsibility
 - Enables division of labor between programmers and designers
 - Facilitates unit testing
 - Easier to understand, change and debug

Advantages of MVC Pattern

Separation of application logic and web design through the *MVC pattern*

- ◆ Integration with template languages
- ◆ Some MVC frameworks provide built-in components

- ◆ Other advantages include
 - Form validation
 - Error handling
 - Request parameter type conversion
 - Internationalization
 - IDE integration

Exploring Spring's Web MVC Framework

- ◆ The Spring Web MVC framework is built on generic Servlet known as a DispatcherServlet class ([Front Controller](#)).
- ◆ The DispatcherServlet class sends the request to the handlers with configurable handler mappings, theme resolution, locale and view resolution along with file uploading.
- ◆ The handleRequest(request, response) method is given by the default handler called Controller interface.
- ◆ The application controller implementation classes of the controller interface are as follows:
 - AbstractController
 - AbstractCommandController
 - SimpleFormController.

Spring MVC Features

- ◆ Powerful configuration of both framework and application classes.
- ◆ Separation of roles.
- ◆ Flexibility in choosing subclasses.
- ◆ Model transfer flexibility.
- ◆ No need of duplication of code.
- ◆ Specific validation and binding.
- ◆ Specific local and theme resolution.
- ◆ Facility of JSP form tag library.

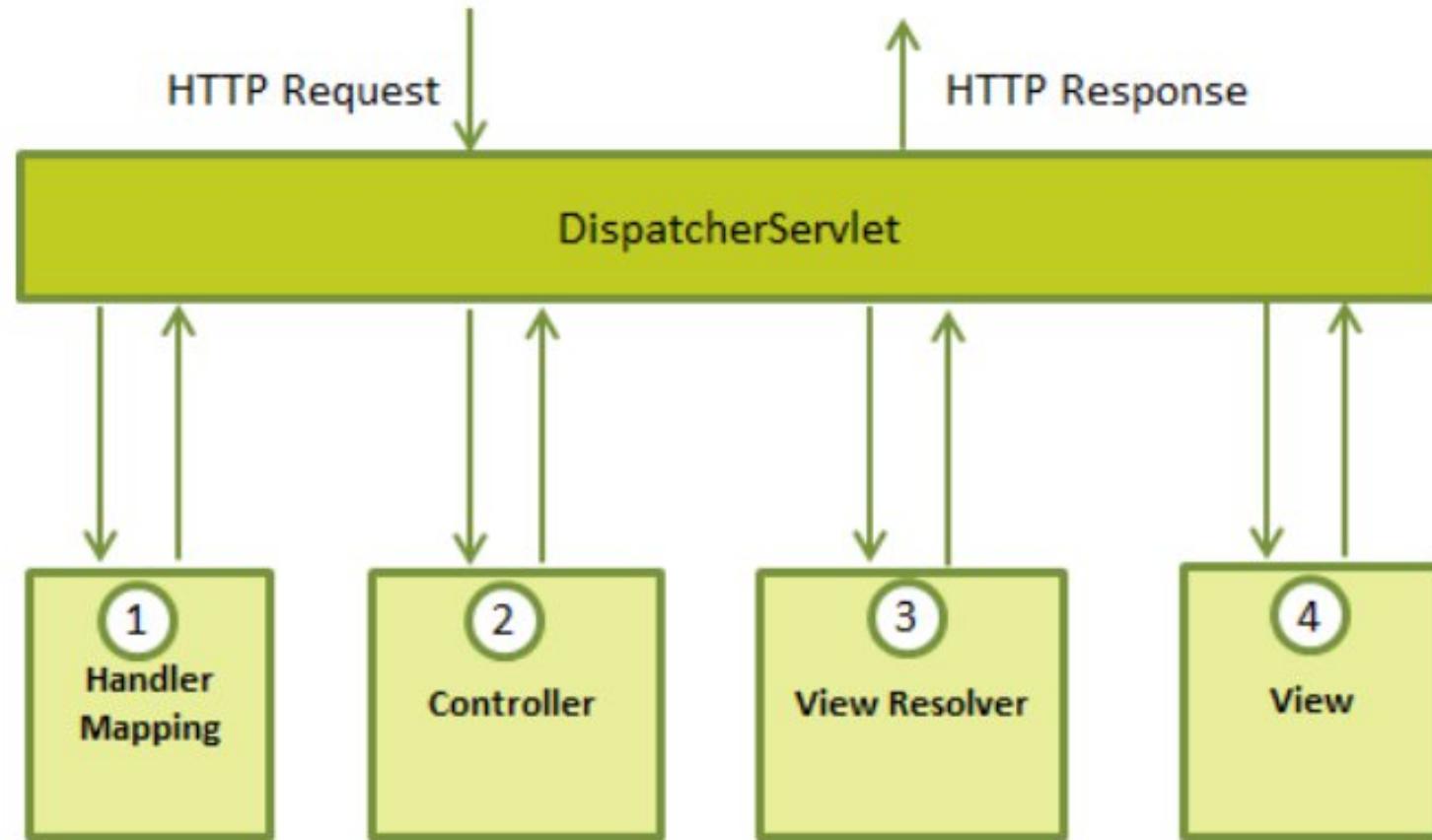
Spring Web MVC

- ◆ Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning.
- ◆ The formal name, "Spring Web MVC," comes from the name of its source module (`spring-webmvc`), but it is more commonly known as "Spring MVC".

Spring Web MVC

- ◆ DispatcherServlet
- ◆ Filters
- ◆ Annotated Controllers
- ◆ Functional Endpoints
- ◆ URI Links
- ◆ Asynchronous Requests
- ◆ CORS
- ◆ Error Responses
- ◆ Web Security
- ◆ HTTP Caching
- ◆ View Technologies
- ◆ MVC Config
- ◆ HTTP/2

DispatcherServlet



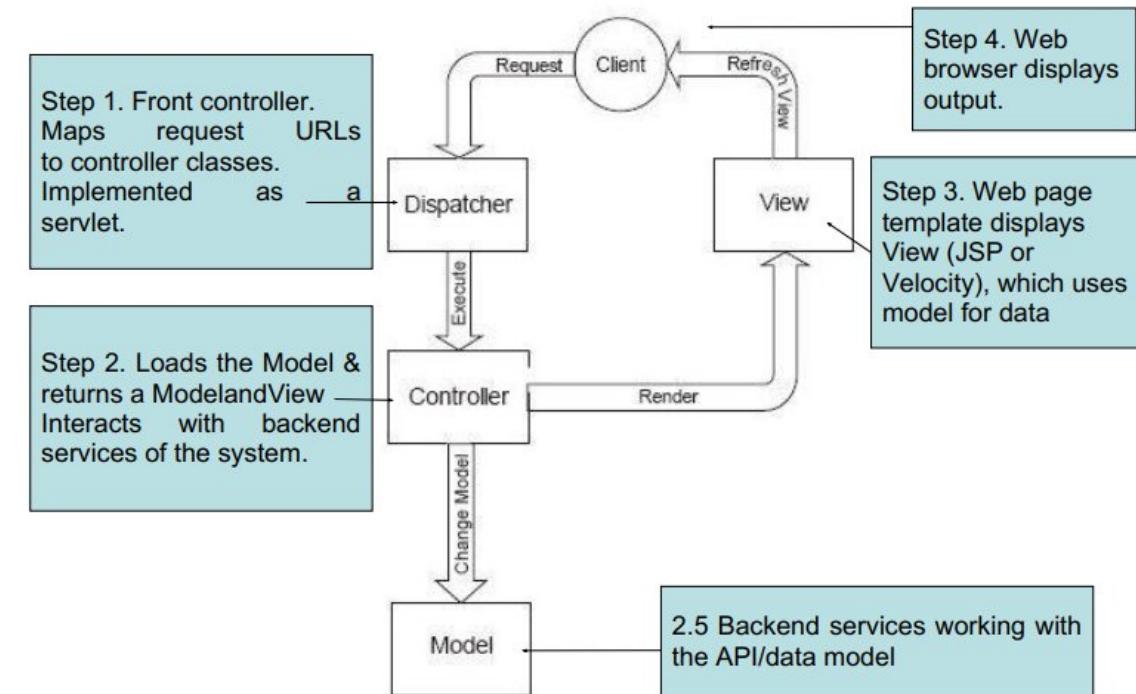
DispatcherServlet

Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet*

- After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
- The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
- The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
- Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

DispatcherServlet

- HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext* which is an extension of the plain *ApplicationContext* with some extra features necessary for Web applications.



DispatcherServlet

- The DispatcherServlet class is important part of spring Web MVC framework.
- It is used for dispatching the request to application controllers.
- The DispatcherServlet class is configured in web.xml file of a web application.
- Map the request using Uniform Resource Locator (URL) mapping in the same web.xml file to handle any request

org.springframework.web.servlet

Class DispatcherServlet

java.lang.Object

javax.servlet.GenericServlet

javax.servlet.http.HttpServlet

org.springframework.web.servlet.HttpServletBean

org.springframework.web.servlet.FrameworkServlet

org.springframework.web.servlet.DispatcherServlet

HadlerMapping/Controller/ViewResolver

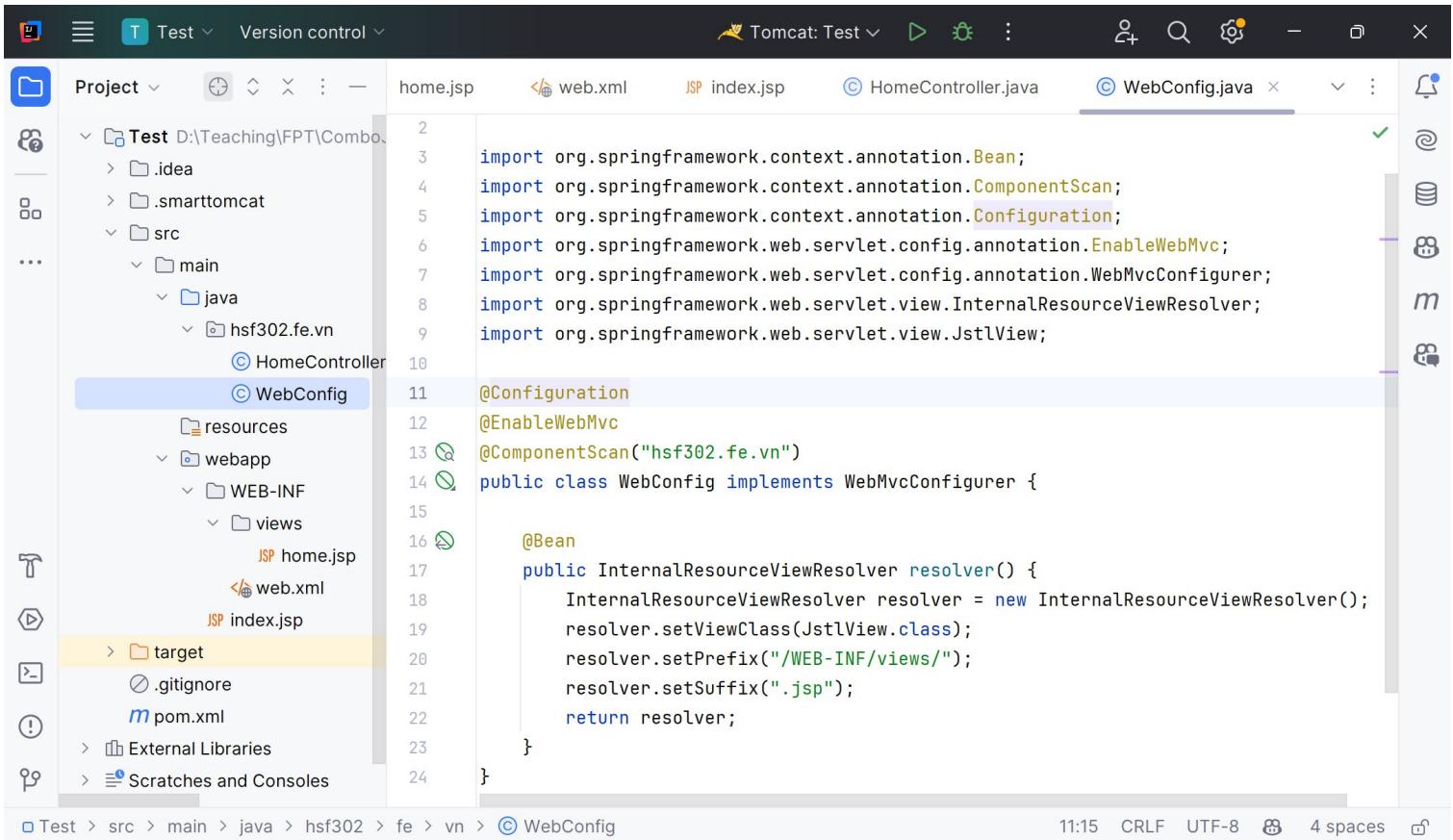
- ◆ Handler mapping: Manages the execution of controllers, provided they match the specified criteria.
- ◆ Controller: It handles the client's request.
- ◆ View resolver: Resolves view names to view used by the DispatcherServlet.
 - The mapping between the Logical name and the Physical View Location is taken care by the View Resolver object.
 - Spring comes with a set of Built-In Spring Resolvers.
 - We can write Custom View Resolvers by implementing the [`org.springframework.web.servlet.ViewResolver`](#) interface

View Resolver

- ◆ **InternalResourceViewResolver**
- ◆ **ContentNegotiatingViewResolver**
- ◆ BeanNameViewResolver
- ◆ FreeMarkerViewResolver
- ◆ JasperReportsViewResolver
- ◆ ResourceBundleViewResolver
- ◆ UrlBasedViewResolver
- ◆ VelocityLayoutViewResolver
- ◆ VelocityViewResolver
- ◆ XmlViewResolver
- ◆ XsltViewResolver

View Resolver

◆ InternalResourceViewResolver



The screenshot shows the IntelliJ IDEA interface with the code for the `WebConfig.java` file. The code defines a class `WebConfig` that implements `WebMvcConfigurer`. It uses `@Configuration`, `@EnableWebMvc`, and `@ComponentScan("hsf302.fe.vn")` annotations. The `resolver()` method returns an `InternalResourceViewResolver` instance with `setViewClass(JstlView.class)`, `setPrefix("/WEB-INF/views/")`, and `setSuffix(".jsp")`.

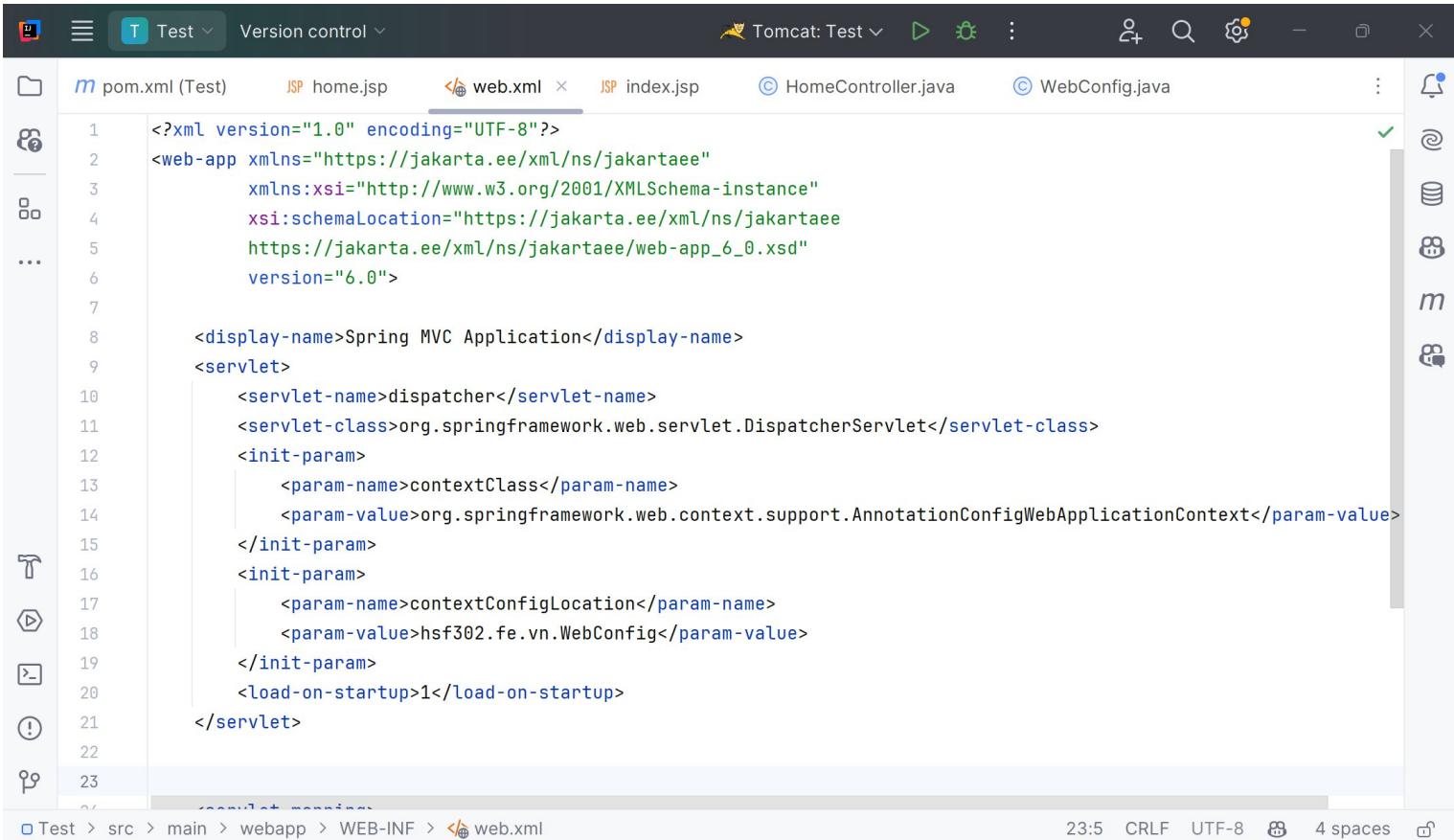
```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan("hsf302.fe.vn")
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public InternalResourceViewResolver resolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setViewClass(JstlView.class);
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```

Configure DispatcherServlet in web.xml

- Spring application will automatically be searched for and loaded by Spring for us.



The screenshot shows a code editor with the following configuration in the `web.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
          https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
          version="6.0">

    <display-name>Spring MVC Application</display-name>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>hsf302.fe.vn.WebConfig</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

```

The code editor interface includes tabs for `pom.xml (Test)`, `home.jsp`, `index.jsp`, `HomeController.java`, and `WebConfig.java`. The `web.xml` tab is active. The status bar at the bottom shows the file path as `Test > src > main > webapp > WEB-INF > web.xml` and the file statistics as `23:5 CRLF UTF-8 4 spaces`.

Overriding *DispatcherServlet* defaults

Following 2 basic ones:

- *InternalResourceViewResolver* (for jsp, css, images etc)
- *ContentNegotiatingViewResolver* (for ContentType response, useful for REST APIs)
- If the Controller returns “**index**”, InternalResourceViewResolver tries to find file as view **/WEB-INF/views/index.jsp**

Controllers

- ◆ The Controllers are central components of MVC
- ◆ Simply add **@Controller** annotation to a class
- ◆ Use **@RequestMapping** to map methods to url

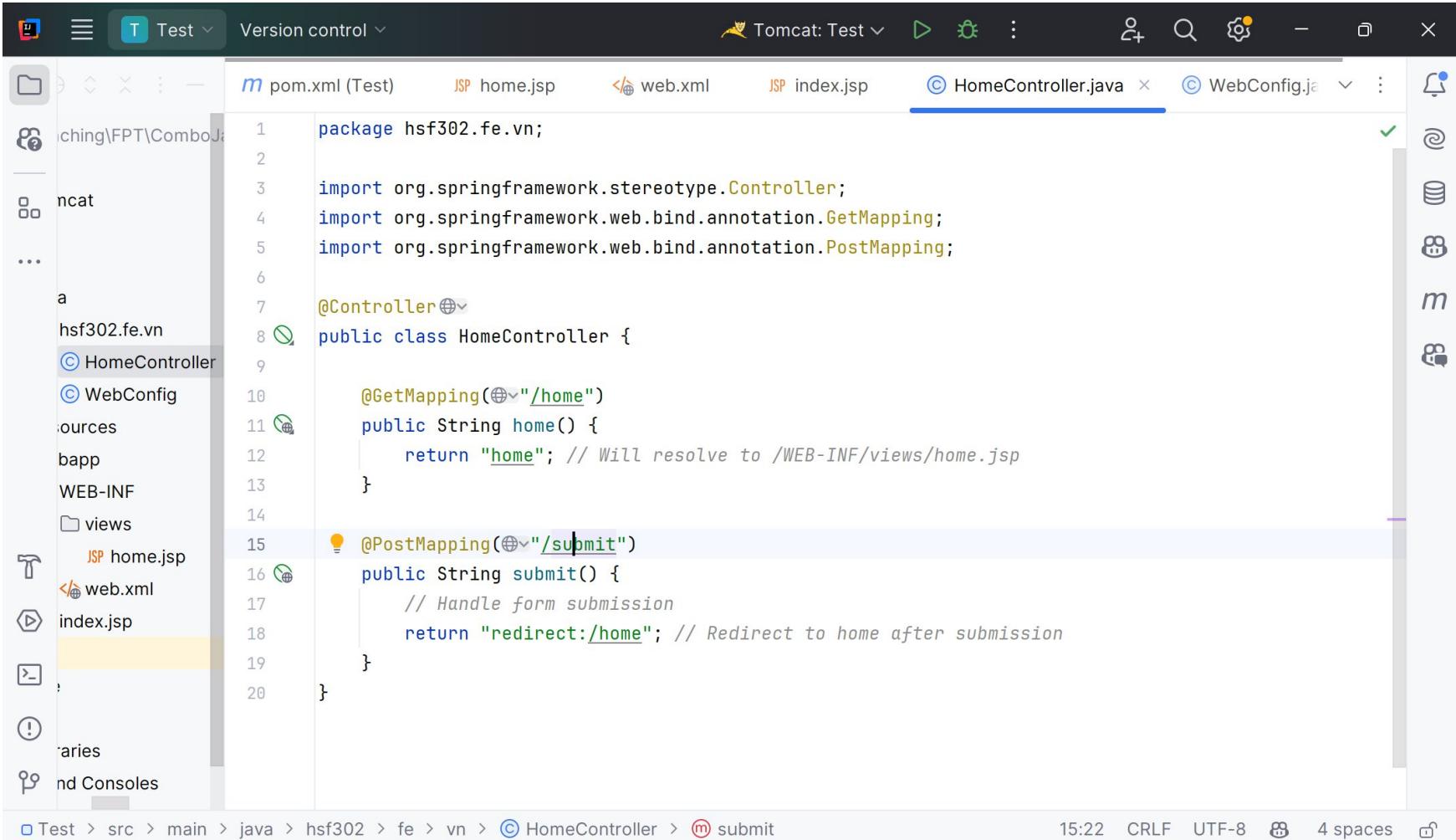
```
@Controller  
public class BaseController {  
  
    @RequestMapping(value="/")  
    public String welcome(ModelMap model) {  
  
        model.addAttribute("message", "Whaddap!!");  
  
        //Spring uses InternalResourceViewResolver and return back index.jsp  
        return "index";  
    }  
    ...  
}
```

Controllers

Controller Annotation

- The `@Controller` annotation defines the class as a Spring MVC controller.
- The `@RequestMapping` annotation is used to map URLs like '/hello' onto an entire class or a particular handler method.
- `@RequestMapping(method = RequestMethod.GET)` is used to declare the `printHello()` method as the controller's default service method to handle HTTP GET request.

Example



The screenshot shows an IDE interface with the following details:

- Toolbar:** Includes icons for file operations, test, version control, Tomcat, and search.
- Project Explorer:** Shows the project structure with files like pom.xml, home.jsp, web.xml, index.jsp, HomeController.java, and WebConfig.java.
- Code Editor:** The active tab is HomeController.java, displaying the following code:

```
package hsf302.fe.vn;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;

@Controller
public class HomeController {

    @GetMapping("/home")
    public String home() {
        return "home"; // Will resolve to /WEB-INF/views/home.jsp
    }

    @PostMapping("/submit")
    public String submit() {
        // Handle form submission
        return "redirect:/home"; // Redirect to home after submission
    }
}
```

The code uses Spring annotations to map URLs to methods and handle form submissions.

Bottom Status Bar: Shows the path as Test > src > main > java > hsf302 > fe > vn > HomeController > submit, and status information like 15:22, CRLF, UTF-8, 4 spaces, and a refresh icon.

Controller arguments

- ◆ **@RequestParam, @PathVariable**
- ◆ **BindingResult** – to access the results of binding and validation (POJO – arbitrary java object that gets populated with request values **@Valid** – to enforce validation of the POJO)
- ◆ **Model, ModelMap, Map<String, ?>** access to the model object
- ◆ Raw **HttpServletRequest, HttpServletResponse, HttpSession**
- ◆ **@RequestHeader, @RequestBody**

More detailed Url Mapping

- @RequestMapping also accepts the following parameters:
 - method (GET/POST/PUT/DELETE...)
 - produces (mimeType)
 - consumes (mimeType)
 - params
 - headers

```
...
@RequestMapping(value = "/", method = RequestMethod.GET, produces = "text/html")
public String welcome(ModelMap model) {

    model.addAttribute("message", "Whaddap!!");

    //Spring uses InternalResourceViewResolver and return back index.jsp
    return "index";
}
...
```

Url Templates in Controllers

- ◆ **@PathVariable** - to map variables in URL paths
- ◆ path variables can also be Regular Expressions
- ◆ Can also do as follows `/message/*/user/{name}`
- ◆ Can also use comma-separated URL parameters (also called Matrix-Variables)
 - To do this, make `setRemoveSemicolonContent=false` for *RequestMappingHandlerMapping*

```
// GET = /message/lars;friends=bob,rob,andy
@RequestMapping(value="/message/{name}", method = RequestMethod.GET)
public String welcome( @PathVariable String name, @MatrixVariable String[] friends, ModelMap model) {

    model.addAttribute("message", "Hello " + name + " from " + friends[0] + " & " + friends[1]);
    //Spring uses InternalResourceViewResolver and return back index.jsp
    return "index";
}
```

Convention over Configuration

GET /student/listAll

```
@Controller
```

```
public class StudentController {
```

```
    @RequestMapping
```

```
    public void listAll(Model model){
```

```
        model.addAttribute(new Student("Test",23));
```

```
}
```

```
}
```

Mapping of class and method name

Model key generated from object type

View with method name selected from request path

Convention over Configuration

- ◆ Mapping By Convention

```
@Controller
```

```
public class StudentController {
```

URL → /student/showList , View → showList.jsp

```
@RequestMapping
```

```
public void showList(Model model){}
```

URL → /student/getStudent, View → getStudent.jsp

```
@RequestMapping
```

```
public Student getStudent(){
```

```
...
```

```
return stu;
```

```
}
```

```
}
```

Model

- Controllers and view share a Java object referred as model, ('M' in MVC)
- A model can be of the type *Model* or can be a *Map* that can represent the model.
- The view uses this to display dynamic data that has been given by the controller

```
// Controller
@RequestMapping(value = "/{name}", method = RequestMethod.GET)
public String welcome(@PathVariable String name, ModelMap model) {
    model.addAttribute("message", "Hello " + name);
    return "index";
}
```

- In View:

```
<html>
    <body><h1>${message}</h1></body>
</html>
```

ModelAndView object in Spring MVC

- ◆ ModelAndView is an object that holds both the model and view. The handler returns the ModelAndView object and DispatcherServlet resolves the view using View Resolvers and View.
- ◆ The View is an object which contains view name in the form of the String and model is a map to add multiple objects.

```
ModelAndView model = new ModelAndView("employeeDetails");
model.addObject("employeeObj", new EmployeeBean(123));
model.addObject("msg", "Employee information.");
return model;
```

@ModelAttribute from Controller

- ◆ You can also use **@ModelAttribute** in controller to directly load URL value into the model
- ◆ A Model can represent objects that can be retrieved from database or files as well
- ◆ Model should not have logic, rather the controller should get the model and “transform” the model based on the request, while sending it to the View

View

- ◆ Spring MVC integrates with many view technologies:
 - JSP
 - Thymeleaf
 - Velocity
 - Freemarker
 - JasperReports
- ◆ Values sent to controller with POST or GET as usual
- ◆ Values made available to the view by the controller

Spring's form tag library

- ◆ Binding-aware JSP tags for handling form elements
- ◆ Integrated with Spring MVC to give the tags access to the model object and reference data
- ◆ Comes from spring-webmvc.jar
- ◆ Add the following to make the tags available:
 - ◆ <%@ taglib prefix="**form**"
uri="<http://www.springframework.org/tags/form>" %>

Spring's form tag library

Tag	Description
form:form	Generates the HTML <form> tag. It has the name attribute that specifies the command object that the inner tags should bind to.
form:input	Represents the HTML input text tag.
form:password	Represents the HTML input password tag.
form:radiobutton	Represents the HTML input radio button tag.
form:checkbox	Represents the HTML input checkbox tag.
form:select	Represents the HTML select list tag.
form:options	Represents the HTML options tag.
form:errors	Represents the HTML span tag. It also generates span tag from the error created as a result of validations.

The attributes: *path* & *modelAttribute/commandName*

- ◆ **commandName/modelAttribute:** name of a variable in the request scope or session scope that contains the information about this form, it should be a bean.
- ◆ **path:** name of a bean property that should be accessed in order to pass the information to from and to the controller.

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
<title>Spring MVC Form Handling</title>
</head>

<body>
    <h2>Employee</h2>
    <form:form method="POST" action="addEmployee" modelAttribute="employee">
        Id: <form:input path="id" />
        <br/>
        Name: <form:input path="name" />
        <input type="submit" value="Submit" />
    </form:form>
</body>
</html>
```

The *form* tag

- ◆ Renders a form tag and exposes the binding to inner tags
- ◆ You can specify any HTML attributes that are valid for an HTML form
- ◆ You can also tell it what the form backing object is (it uses ‘**command**’ by default).

```
@RequestMapping("/person/add")
public String addPerson(Model model) {
    Person person = new Person();
    model.addAttribute(person);
    return "addPerson";
}
```

```
<form:form method="get" commandName="person">
    <form:input path="name" />
</form:form>
```

```
<form method="get">
    <input type="text" name="name" />
</form>
```

The *input* tag

- ◆ Renders an HTML input tag with a type of 'text'
- ◆ You can specify any HTML attributes valid for an HTML input
- ◆ You bind it to your model object by specifying the path relative to the backing object

```
@RequestMapping("/person/add")
public String addPerson(Model model) {
    Person person = new Person();
    person.setName("Spencer");
    model.addAttribute(person);
    return "addPerson";
}
```

```
<form:form method="get" commandName="person">
    <form:input path="name" />
</form:form>
```

```
<form method="get">
    <input type="text" name="name" value="Spencer" />
</form>
```

The *checkbox* tag

- ◆ Renders an HTML input tag with a type of 'checkbox'

```
public class Person {  
    private boolean admin;  
    private String[] languages;  
}
```

```
<form:form commandName="person">  
    <form:checkbox path="admin" />  
    <form:checkbox path="languages" value="Java" />  
    <form:checkbox path="languages" value="Scala" />  
</form:form>
```

```
<form>  
    <input type="checkbox" name="admin" value="true" />  
    <input type="checkbox" name="languages" value="Java" />  
    <input type="checkbox" name="languages" value="Scala" />  
</form>
```

The *checkboxes* tag

- ◆ Similar to *checkbox* tag, but creates multiple checkboxes instead of one

```
public class Person {  
    private boolean admin;  
    private String[] favoriteLanguages;  
    private List<String> allLanguages;  
}
```

```
<form:form commandName="person">  
    <form:checkbox path="admin" />  
    <form:checkboxes path="favoriteLanguages" items="${allLanguages}" />  
</form:form>
```

```
<form>  
    <input type="checkbox" name="admin" value="true" />  
    <input type="checkbox" name="favoriteLanguages" value="Java" />  
    <input type="checkbox" name="favoriteLanguages" value="Scala" />  
</form>
```

The *password* tag

- ◆ Renders an HTML input tag with a type of ‘password’

```
<form:form>
    <form:input path="username" />
    <form:password path="password" />
</form:form>
```

```
<form>
    <input type="text" name="username" />
    <input type="password" name="password" />
</form>
```

The **select** tag

- ◆ Renders an HTML select tag
- ◆ It can figure out whether multiple selections should be allowed
- ◆ You can bind options using this tag, as well as by nesting *option* and *options* tags

```
<form:select path="favoriteLanguage" items="${allLanguages}" />
```

```
<select name="favoriteLanguage">
    <option value="Java">Java</option>
    <option value="Scala">Scala</option>
</form>
```

The *option/options* tags

- ◆ Renders an HTML option (or multiple options)
- ◆ Nested within a select tag
- ◆ Renders ‘selected’ based on the value bound to the select tag

```
<form:select path="favoriteLanguage">
    <form:option value="3" label=".NET" />
    <form:options items="${languages}" itemValue="id" itemLabel="name"
/>
</form:select>
```

```
<select name="favoriteLanguage">
    <option value=".NET">.NET</option>
    <option value="Java">Java</option>
    <option value="Scala">Scala</option>
</form>
```

The *errors* tag

- ◆ Renders an HTML span tag, containing errors for given fields
- ◆ You specify which fields to show errors for by specifying a path
 - path="name" – Would display errors for name field.
 - path="*" – Would display all errors

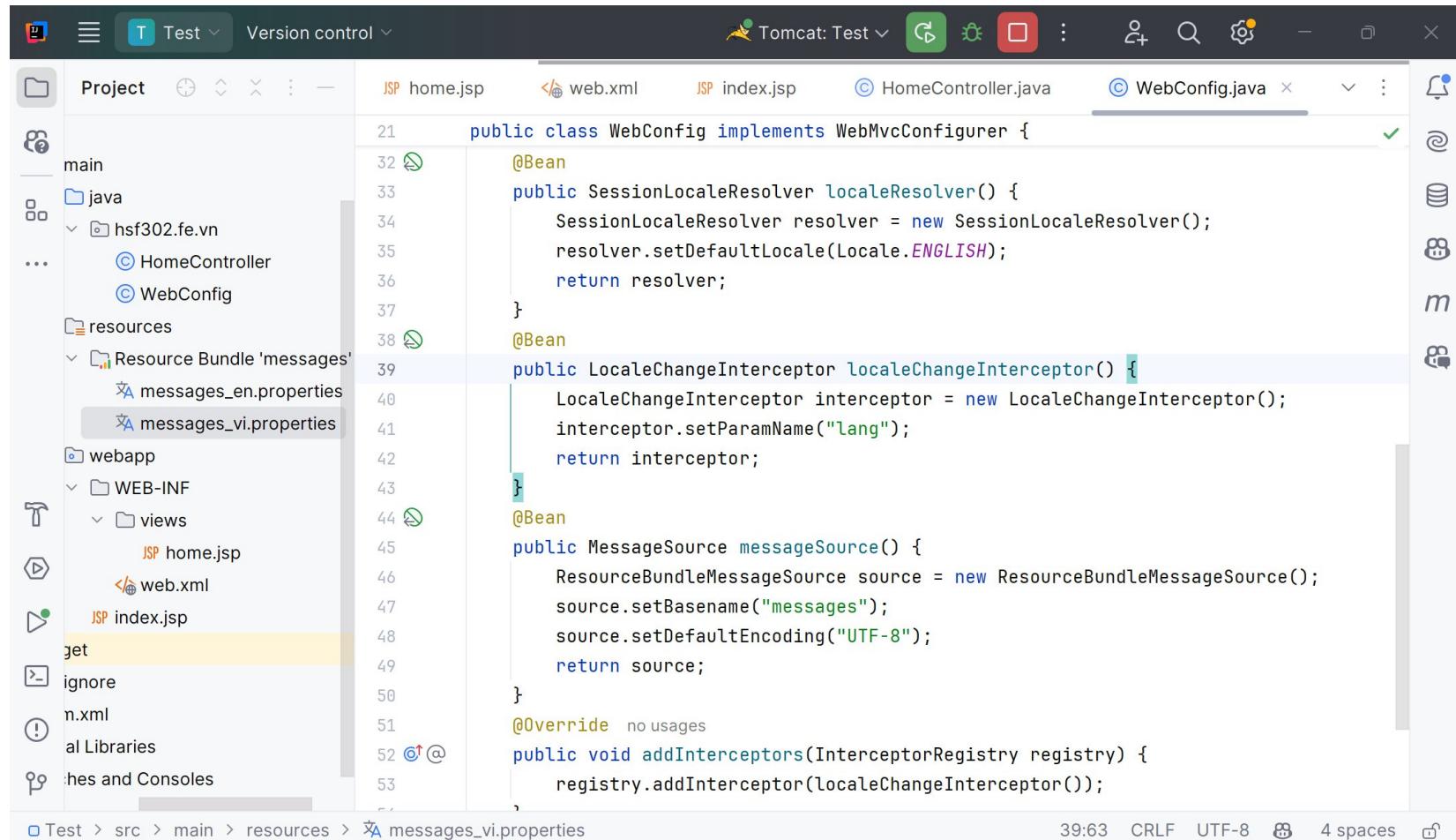
```
<form:errors path="*" cssClass="errorBox" />
<form:errors path="name" cssClass="specificErrorBox" />
```

```
<span name="*.errors" class="errorBox">Name is required.</span>
<span name="name.errors" class="specificErrorBox">Name is required.</span>
```

Spring Exception

- ◆ @ExceptionHandler only handles exception getting raised from the controller where it is defined.
- ◆ It will not handle exceptions getting raised from other controllers. @ControllerAdvice annotation solves this problem.
- ◆ @ControllerAdvice annotation is used to define @ExceptionHandler, @InitBinder, and @ModelAttribute methods that apply to all @RequestMapping methods.

Locales



The screenshot shows an IDE interface with the following details:

- Project Structure:** The project structure on the left includes a main folder containing a java folder with `hsf302.fe.vn`, `HomeController`, and `WebConfig`; a resources folder containing a Resource Bundle 'messages' with `messages_en.properties` and `messages_vi.properties`; and a webapp folder with WEB-INF and views.
- Code Editor:** The code editor displays `WebConfig.java`. The code implements `WebMvcConfigurer` and defines three `@Bean` methods:
 - `SessionLocaleResolver`: Sets the default locale to `Locale.ENGLISH`.
 - `LocaleChangeInterceptor`: Sets the parameter name to `"lang"`.
 - `MessageSource`: Sets the basename to `"messages"` and the default encoding to `"UTF-8"`.
- Status Bar:** The status bar at the bottom shows the file path as `Test > src > main > resources > messages_vi.properties`, and the file statistics as 39:63 CRLF UTF-8 4 spaces.

Locales

The screenshot shows a Java development environment with the following details:

- Project Structure:** The project is named "Test". It contains a "main" folder, a "java" folder with "hsf302.fe.vn" and "HomeController", "WebConfig" classes, and a "resources" folder containing "messages" and "WEB-INF" subfolders. "messages" contains "messages_en.properties" and "messages_vi.properties".
- Code Editor:** The "home.jsp" file is open. It includes JSP tags like <%@ page %>, <%@ taglib %>, and <spring:message>. It also includes HTML tags for the page structure.
- Properties File:** The "messages_en.properties" file is shown in the editor, containing the entry "welcome=Welcome".
- IDE Interface:** The interface includes tabs for "home.jsp", "messages_en.properties", "web.xml", "index.jsp", "HomeController.java", and "WebConfig.java". The status bar at the bottom shows the current time as 11:30, encoding as CRLF/UTF-8, and code style settings.

Spring Validation

- ◆ Spring provides a simplified set of APIs and supporting classes for validating domain objects.
- ◆ Spring features a Validator interface that you can use to validate objects. The Validator interface works using an Errors object so that while validating, validators can report validation failures to the Errors object.
- ◆ *Validation using Spring's Validator interface*

```
public interface Validator {  
  
    /** Can this instances of the supplied clazz */  
    boolean supports(Class<?> clazz);  
  
    /**  
     * Validate the supplied target object, which must be  
     * @param target the object that is to be validated  
     * @param errors contextual state about the validation process  
     */  
    void validate(Object target, Errors errors);  
}
```

Spring Validation

- Standard Constraints

Annotation	Type	Description
@Min(10)	Number	must be higher or equal
@Max(10)	Number	must be lower or equal
@AssertTrue	Boolean	must be true, null is valid
@AssertFalse	Boolean	must be false, null is valid
@NotNull	any	must not be null
@NotEmpty	String / Collection's	must be not null or empty
@NotBlank	String	@NotEmpty and whitespaces ignored
@Size(min, max)	String / Collection's	must be between boundaries
@Past	Date / Calendar	must be in the past
@Future	Date / Calendar	must be in the future
@Pattern	String	must math the regular expression



```
import javax.validation.constraints.Max;  
import javax.validation.constraints.Min;  
import javax.validation.constraints.NotNull;  
import javax.validation.constraints.Past;  
import javax.validation.constraints.Size;  
import org.hibernate.validator.constraints.Email;  
import org.hibernate.validator.constraints.NotEmpty;  
import org.springframework.format.annotation.DateTimeFormat;  
@Phone: defined custom validator.
```

```
@Size(min=2, max=30)  
private String name;  
  
@NotEmpty @Email  
private String email;  
  
@NotNull @Min(18) @Max(100)  
private Integer age;  
  
@NotNull  
private Gender gender;  
  
@DateTimeFormat(pattern="MM/dd/yyyy")  
@NotNull @Past  
private Date birthday;  
  
@Phone  
private String phone;
```

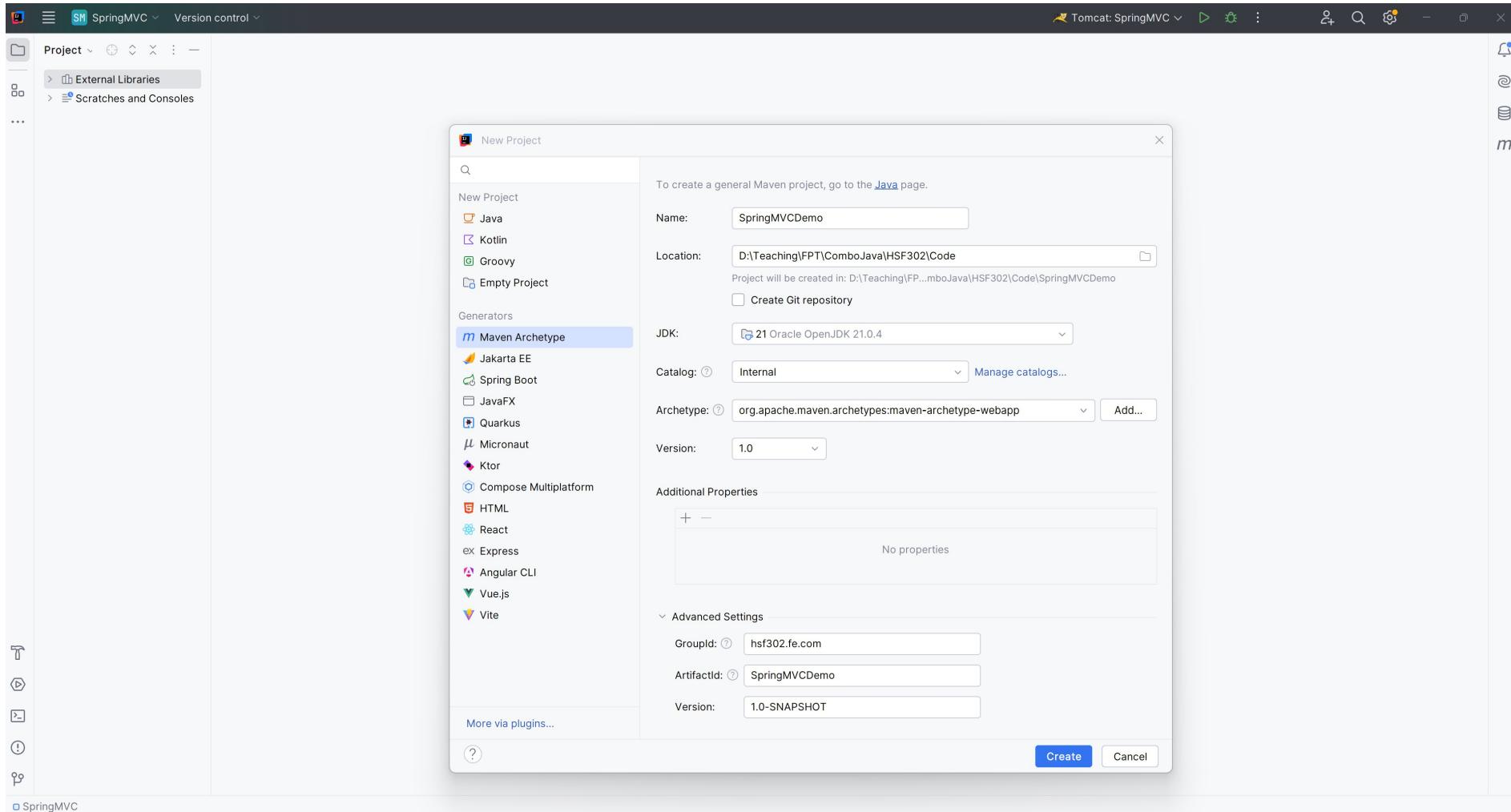
@Autowired

- ◆ @Autowired on Setter Methods
- ◆ @Autowired on Properties
- ◆ @Autowired on Constructors
- ◆ @Autowired on Constructors

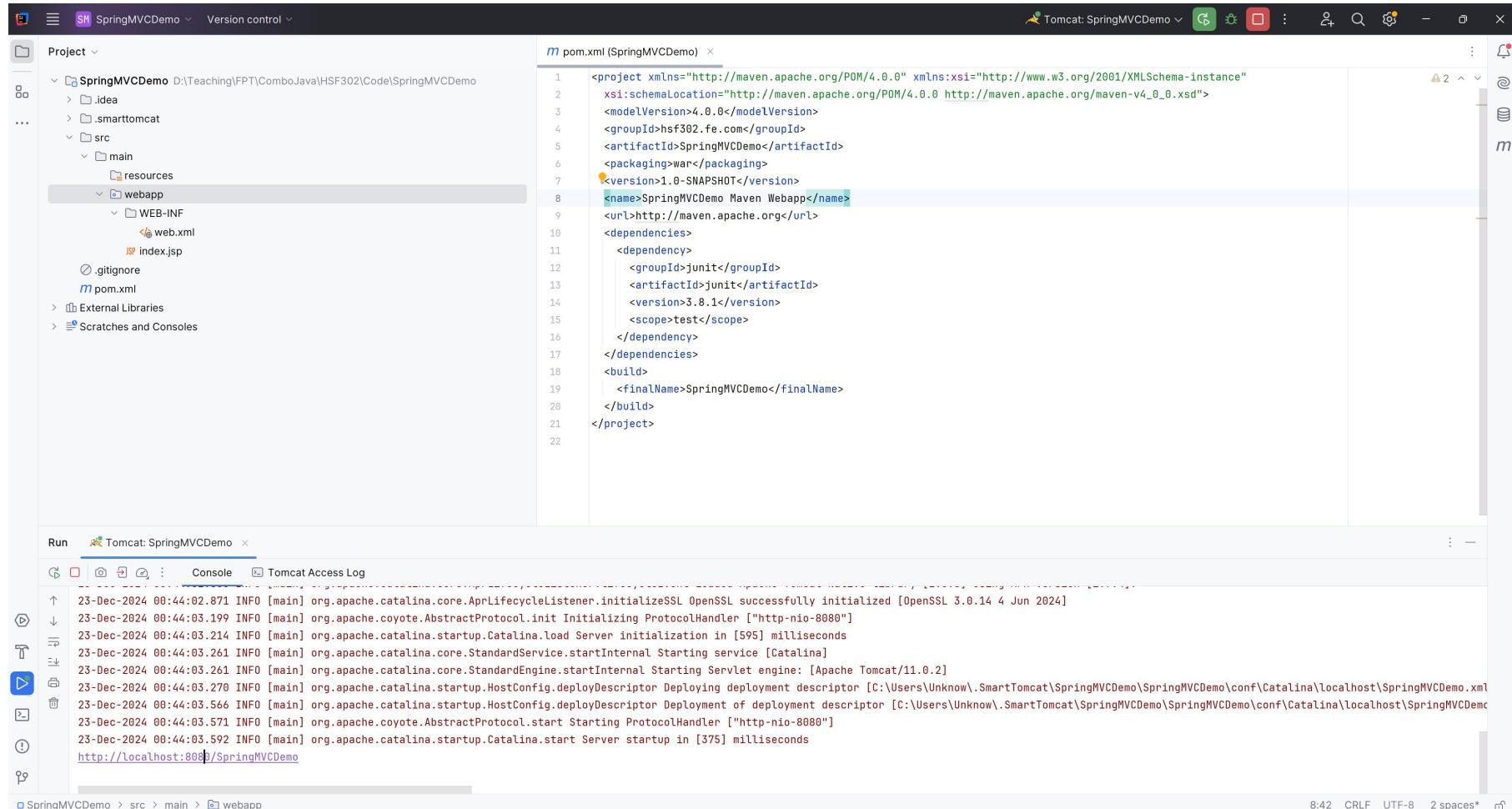
```
public class MyController {  
  
    private MyService myService;  
  
    public MyController(MyService aService) { // constructor based injection  
        this.myService = aService;  
    }  
  
    public void setMyService(MyService aService) { // setter based injection  
        this.myService = aService;  
    }  
  
    @Autowired  
    public void setMyService(MyService aService) { // autowired by Spring  
        this.myService = aService;  
    }  
  
    @RequestMapping("/blah")  
    public String someAction()  
    {  
        // do something here  
        myService.foo();  
  
        return "someView";  
    }  
}
```

Demo SpringMVC

Open IntelliJ, File | New | Maven Project



Run Project



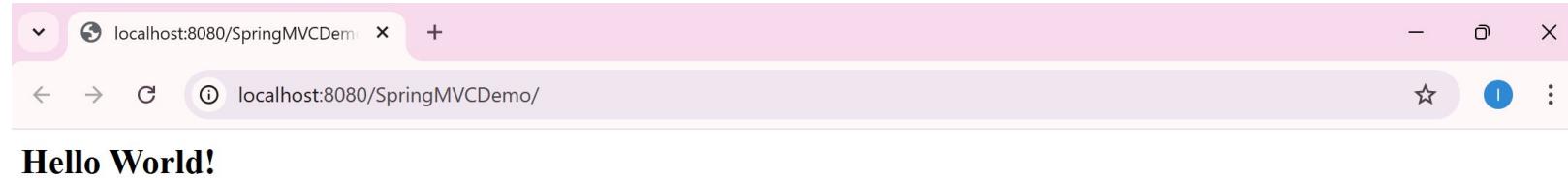
The screenshot shows a Java development environment with the following details:

- Project Structure:** The project is named "SpringMVC Demo" and is located at D:\Teaching\FPT\ComboJava\HSF302\Code\SpringMVC Demo. It contains a "src" folder with a "main" directory containing "resources" and "webapp". The "webapp" folder includes "WEB-INF" and "index.jsp".
- pom.xml Content:**

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>hsf302.fe.com</groupId>
  <artifactId>SpringMVC Demo</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>SpringMVC Demo Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>SpringMVC Demo</finalName>
  </build>
</project>
```
- Run Tab:** The "Tomcat: SpringMVC Demo" tab is active, showing the following log entries:

```
23-Dec-2024 00:44:02.871 INFO [main] org.apache.catalina.core.AprLifecycleListener.initializeSSL OpenSSL successfully initialized [OpenSSL 3.0.14 4 Jun 2024]
23-Dec-2024 00:44:03.199 INFO [main] org.apache.coyote.AbstractProtocol.init Initializing ProtocolHandler ["http-nio-8080"]
23-Dec-2024 00:44:03.214 INFO [main] org.apache.catalina.startup.Catalina.load Server initialization in [595] milliseconds
23-Dec-2024 00:44:03.263 INFO [main] org.apache.catalina.core.StandardService.startInternal Starting service [Catalina]
23-Dec-2024 00:44:03.261 INFO [main] org.apache.catalina.core.StandardEngine.startInternal Starting Servlet engine: [Apache Tomcat/11.0.2]
23-Dec-2024 00:44:03.270 INFO [main] org.apache.catalina.startup.HostConfig.deployDescriptor Deploying deployment descriptor [C:\Users\Unknown\.SmartTomcat\SpringMVC Demo\SpringMVC Demo\conf\Catalina\localhost\SpringMVC Demo.xml]
23-Dec-2024 00:44:03.566 INFO [main] org.apache.catalina.startup.HostConfig.deployDescriptor Deployment of deployment descriptor [C:\Users\Unknown\.SmartTomcat\SpringMVC Demo\SpringMVC Demo\conf\Catalina\localhost\SpringMVC Demo.xml]
23-Dec-2024 00:44:03.573 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]
23-Dec-2024 00:44:03.592 INFO [main] org.apache.catalina.startup.Catalina.start Server startup in [375] milliseconds
http://localhost:8080/SpringMVC Demo
```

Result

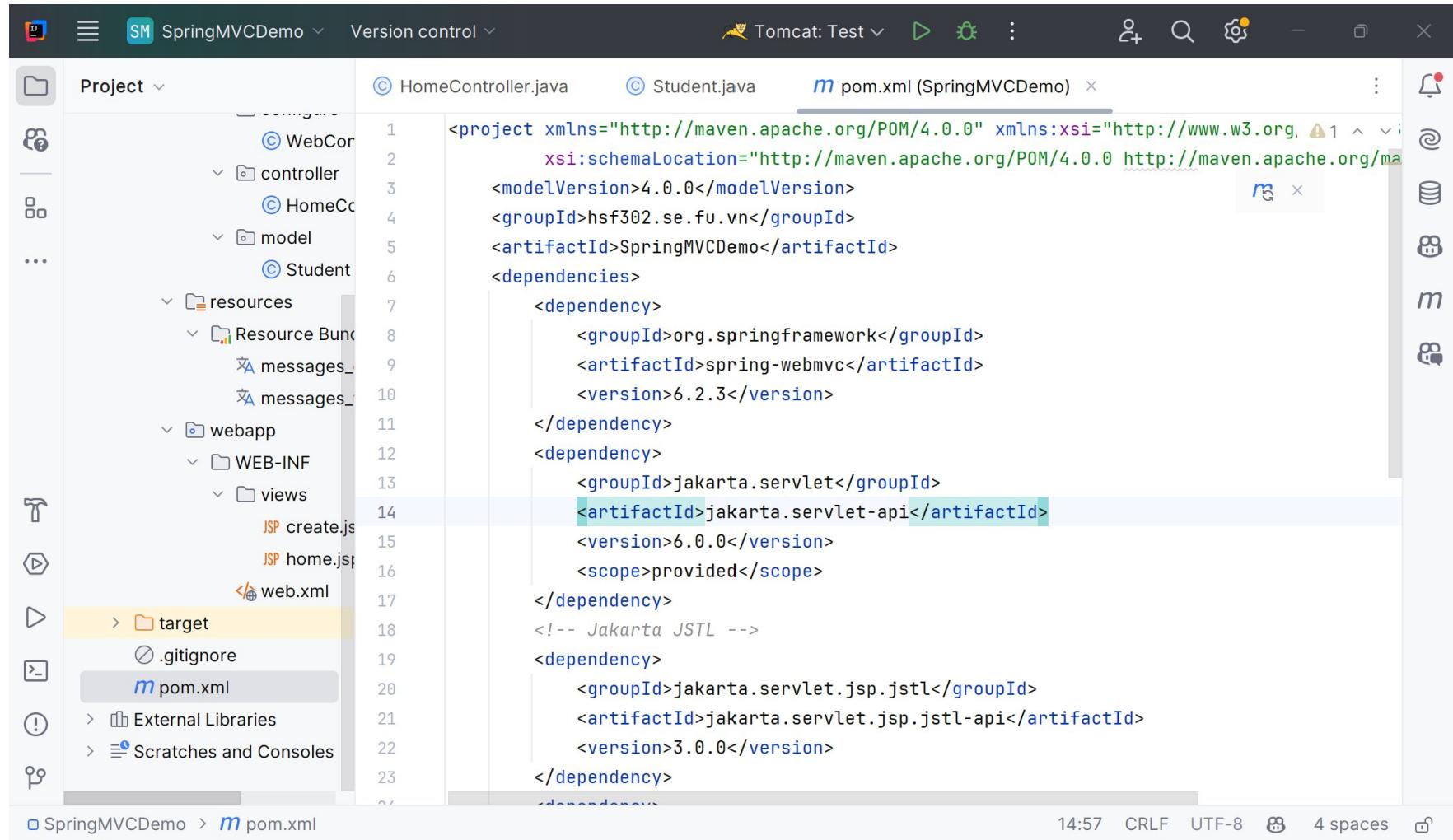


Create a structure of Maven Project

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** On the left, the project structure is displayed under the "Project" tab. It includes the "src" directory with "main" and "resources" sub-directories, and the "webapp" directory which contains "WEB-INF" and "views". Inside "views", there are JSP files named "create.jsp" and "home.jsp", and a "web.xml" file.
- Code Editor:** The main area shows the "Student.java" file content. The code defines a "Student" class with attributes "id", "email", and "password", and their corresponding validation annotations using Jakarta Validation constraints.
- Toolbars and Status Bar:** The top bar includes tabs for "HomeController.java", "Student.java", and "pom.xml (SpringMVCDemo)". The status bar at the bottom shows the file path "SpringMVCDemo > src > main > webapp", and the bottom right corner displays the time as "38:36" and encoding as "CRLF UTF-8 4 spaces".

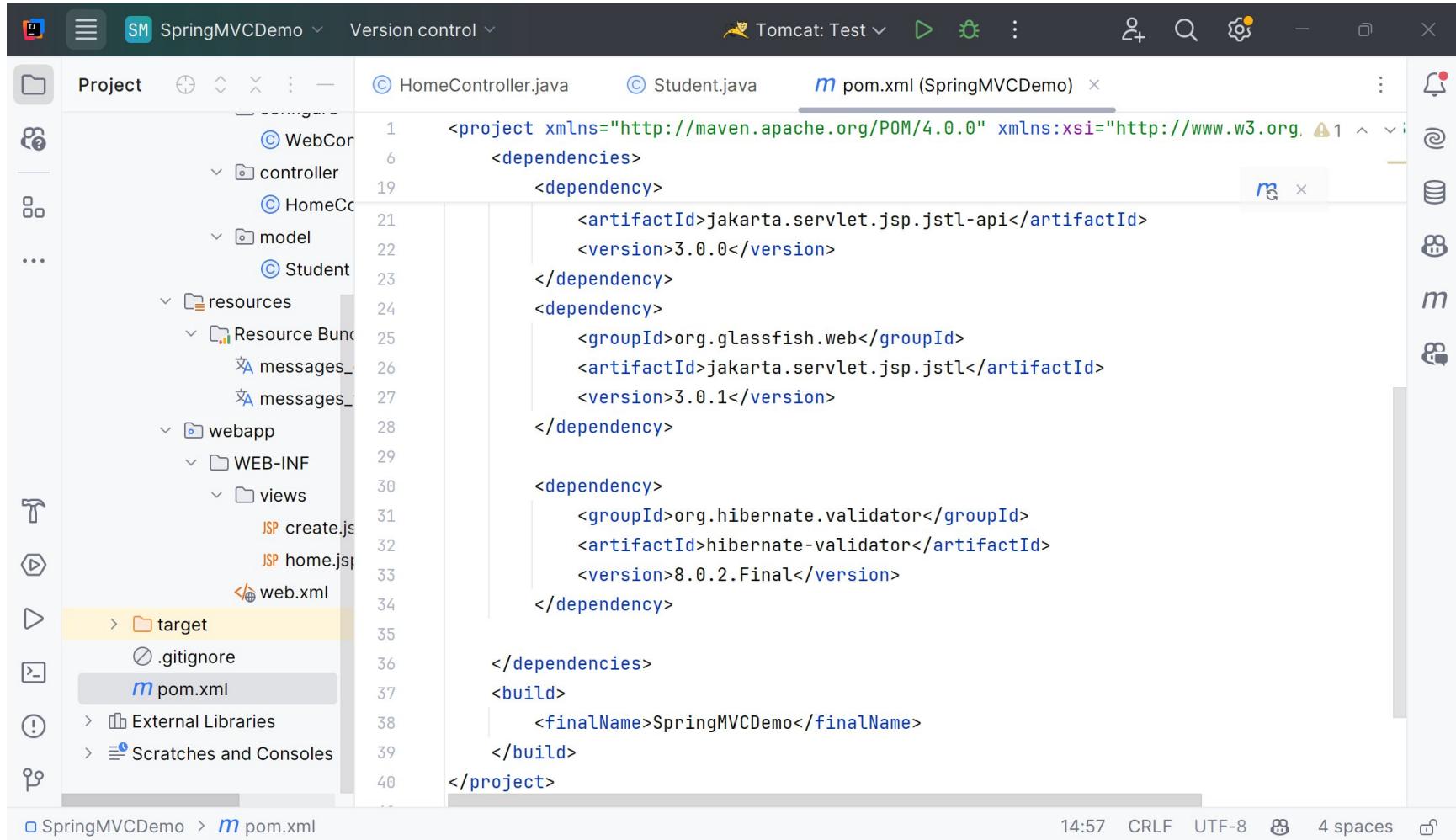
Edit pom.xml



The screenshot shows a code editor interface with the following details:

- Project Structure:** The left sidebar displays the project structure for "SpringMVCDemo". It includes a "controller" package containing "HomeController.java" and "Student.java", a "model" package containing "Student.java", a "resources" folder containing "Resource Bundles" (messages_en.properties, messages_vi.properties), and a "webapp" folder containing "WEB-INF" (with "views" and "create.jsp", "home.jsp") and "target" (containing ".gitignore" and "pom.xml").
- Code Editor:** The main pane shows the content of "pom.xml". The XML code defines a Maven project with group ID "hsf302.se.fu.vn", artifact ID "SpringMVCDemo", and model version 4.0.0. It lists dependencies for "org.springframework" (spring-webmvc) and "jakarta.servlet" (jakarta.servlet-api). It also includes dependencies for "jakarta.servlet.jsp.jstl" (jakarta.servlet.jsp.jstl and jakarta.servlet.jsp.jstl-api).
- Status Bar:** The bottom status bar indicates the file is saved ("SpringMVCDemo > pom.xml"), the current time (14:57), and encoding (CRLF, UTF-8, 4 spaces).

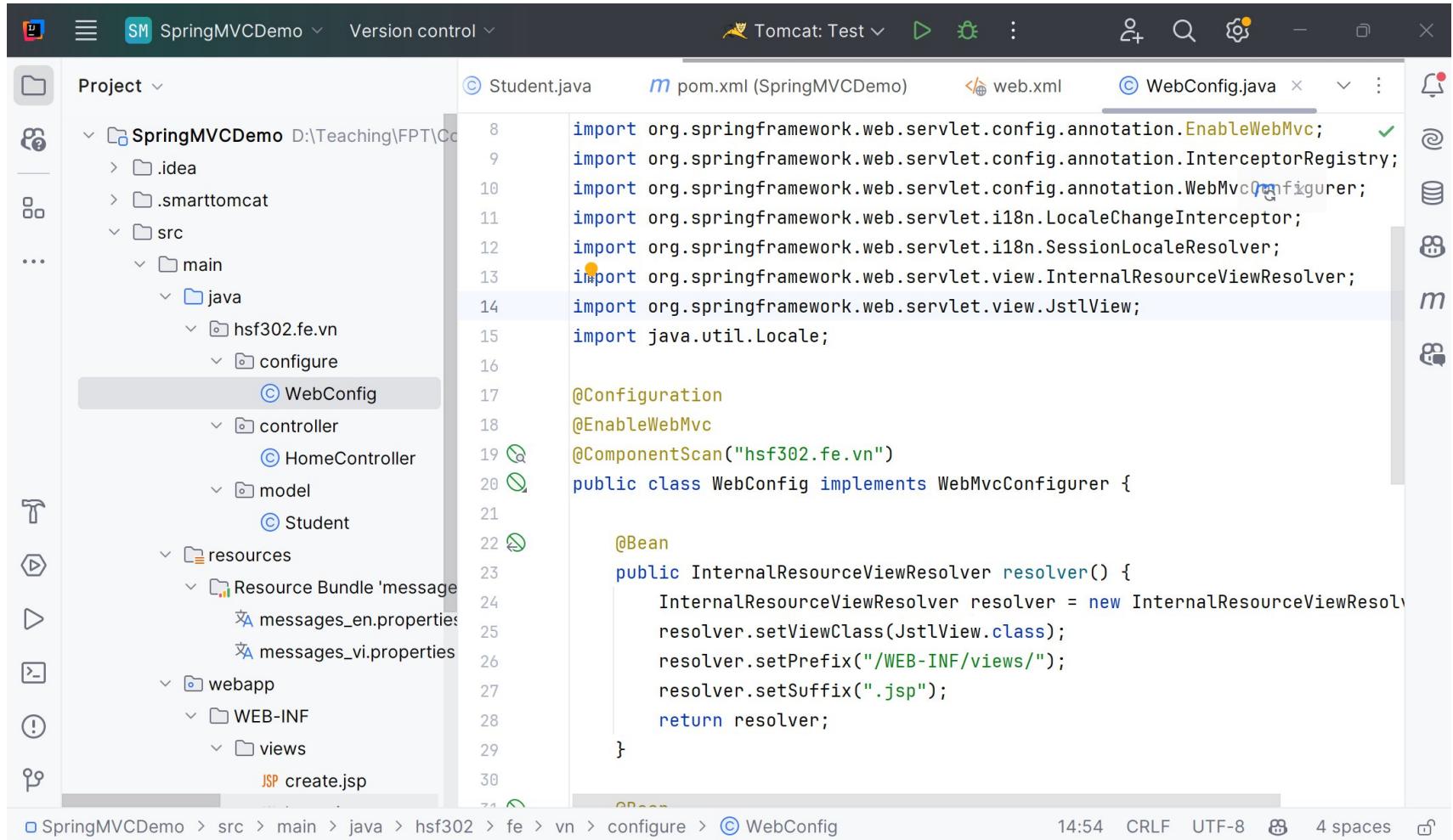
Edit pom.xml



The screenshot shows a code editor window in an IDE, displaying the `pom.xml` file for a project named "SpringMVCDemo". The project structure on the left includes packages for WebCore, controller, model, and Student, along with resources and webapp folders containing JSP files like `create.jsp` and `home.jsp`, and a `web.xml` file. The `pom.xml` file itself is open in the main editor area, showing Maven dependency declarations for Jakarta Servlet API, GlassFish Web, Hibernate Validator, and other components. The file is saved at 14:57 with CRLF line endings.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>SpringMVCDemo</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
            <version>3.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.web</groupId>
            <artifactId>jakarta.servlet.jsp.jstl</artifactId>
            <version>3.0.1</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate.validator</groupId>
            <artifactId>hibernate-validator</artifactId>
            <version>8.0.2.Final</version>
        </dependency>
    </dependencies>
    <build>
        <finalName>SpringMVCDemo</finalName>
    </build>
</project>
```

Create the WebConfig.java

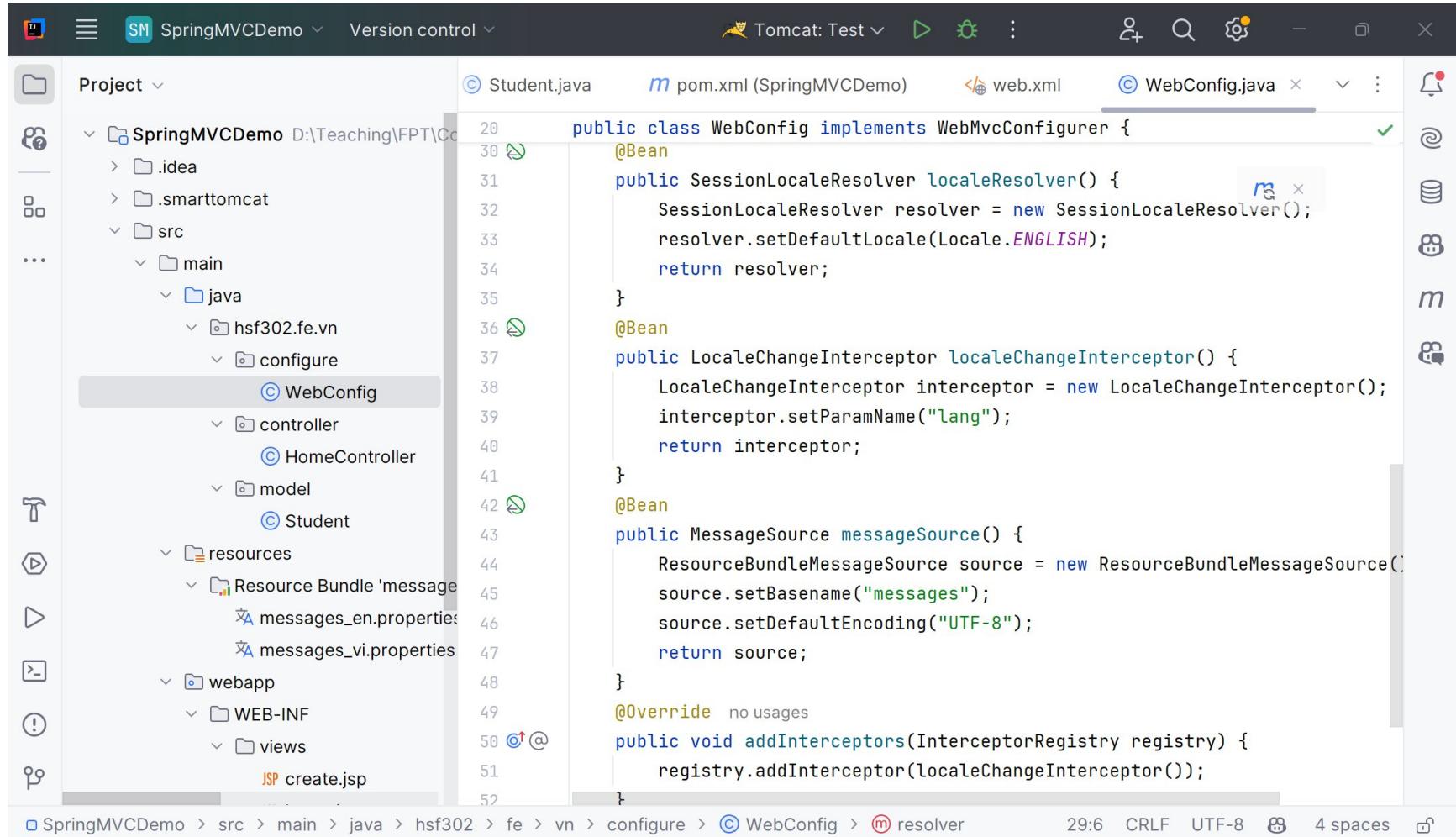


```
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;
import java.util.Locale;

@Configuration
@EnableWebMvc
@ComponentScan("hsf302.fe.vn")
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public InternalResourceViewResolver resolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setViewClass(JstlView.class);
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```

Edit the WebConfig.java



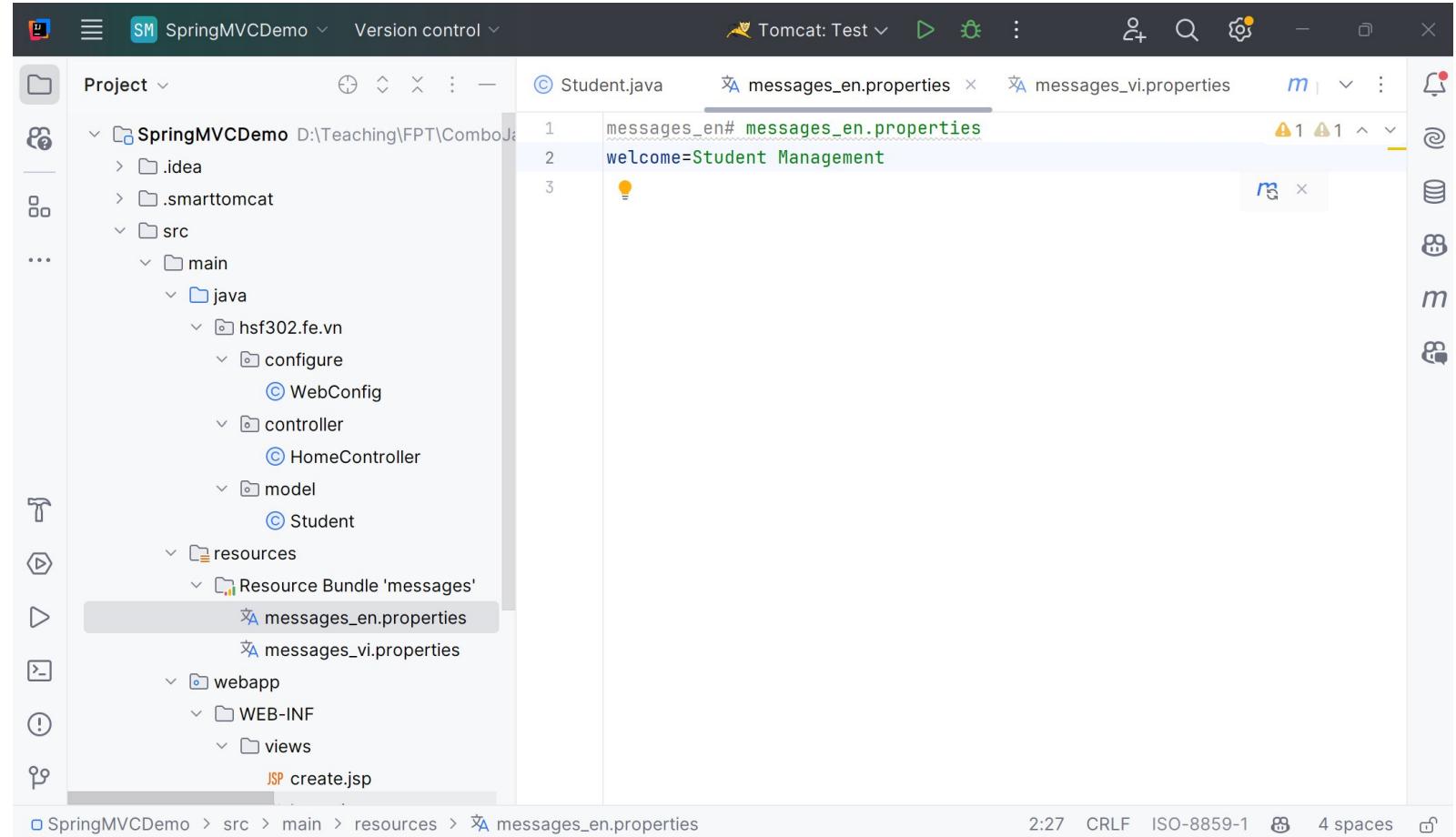
The screenshot shows the IntelliJ IDEA interface with the following details:

- Project:** SpringMVC Demo (D:\Teaching\FPT\Code)
- File:** WebConfig.java
- Code Content:** The code implements the WebMvcConfigurer interface. It defines three @Bean methods: localeResolver, localeChangeInterceptor, and messageSource. The localeResolver returns a SessionLocaleResolver set to ENGLISH. The localeChangeInterceptor sets the param name to "lang". The messageSource uses a ResourceBundleMessageSource with basename "messages" and default encoding UTF-8.

```
public class WebConfig implements WebMvcConfigurer {
    @Bean
    public SessionLocaleResolver localeResolver() {
        SessionLocaleResolver resolver = new SessionLocaleResolver();
        resolver.setDefaultLocale(Locale.ENGLISH);
        return resolver;
    }
    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor interceptor = new LocaleChangeInterceptor();
        interceptor.setParamName("lang");
        return interceptor;
    }
    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource source = new ResourceBundleMessageSource();
        source.setBasename("messages");
        source.setDefaultEncoding("UTF-8");
        return source;
    }
    @Override no usages
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }
}
```

- Toolbars and Status Bar:** The status bar at the bottom shows the file path (SpringMVC Demo > src > main > java > hsf302 > fe > vn > configure > WebConfig.java), line numbers (50 to 52), and file statistics (29:6 CRLF UTF-8 4 spaces).

Create messages_en.properties and messages_vi.properties



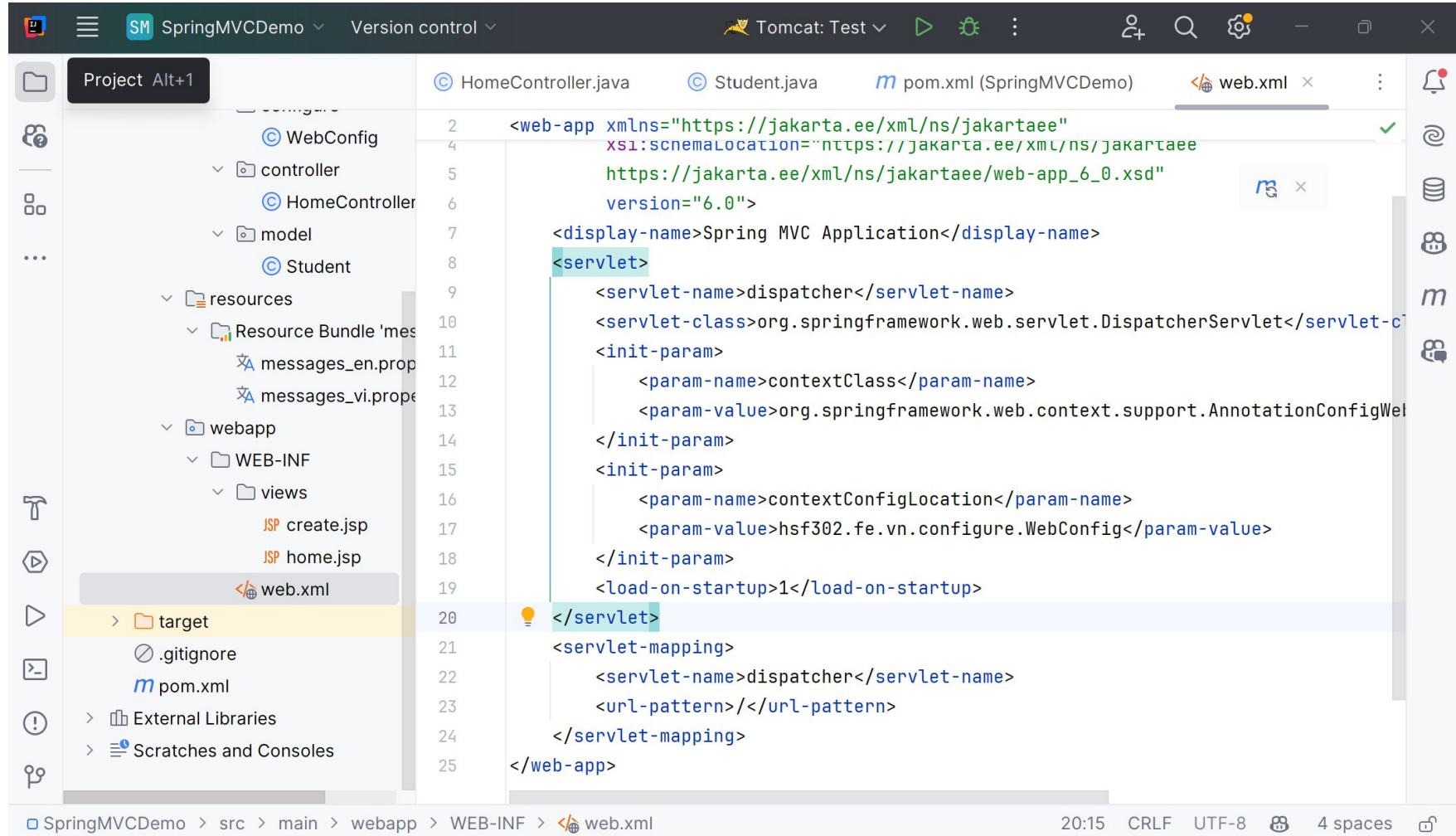
The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The project is named "SpringMVC Demo". It contains a ".idea" folder, a ".smarttomcat" folder, and a "src" directory. The "src" directory is expanded, showing "main", "java", "resources", and "webapp". The "resources" directory contains a "messages" folder which holds "messages_en.properties" and "messages_vi.properties".
- Code Editor:** The "messages_en.properties" file is open in the editor. The code is:

```
1 messages_en# messages_en.properties
2 welcome=Student Management
3
```

A yellow lightbulb icon is visible in the gutter between lines 2 and 3.
- Status Bar:** The status bar at the bottom shows the path "SpringMVC Demo > src > main > resources > messages_en.properties", the time "2:27", the encoding "CRLF", the character set "ISO-8859-1", and the settings "4 spaces".

Edit web.xml



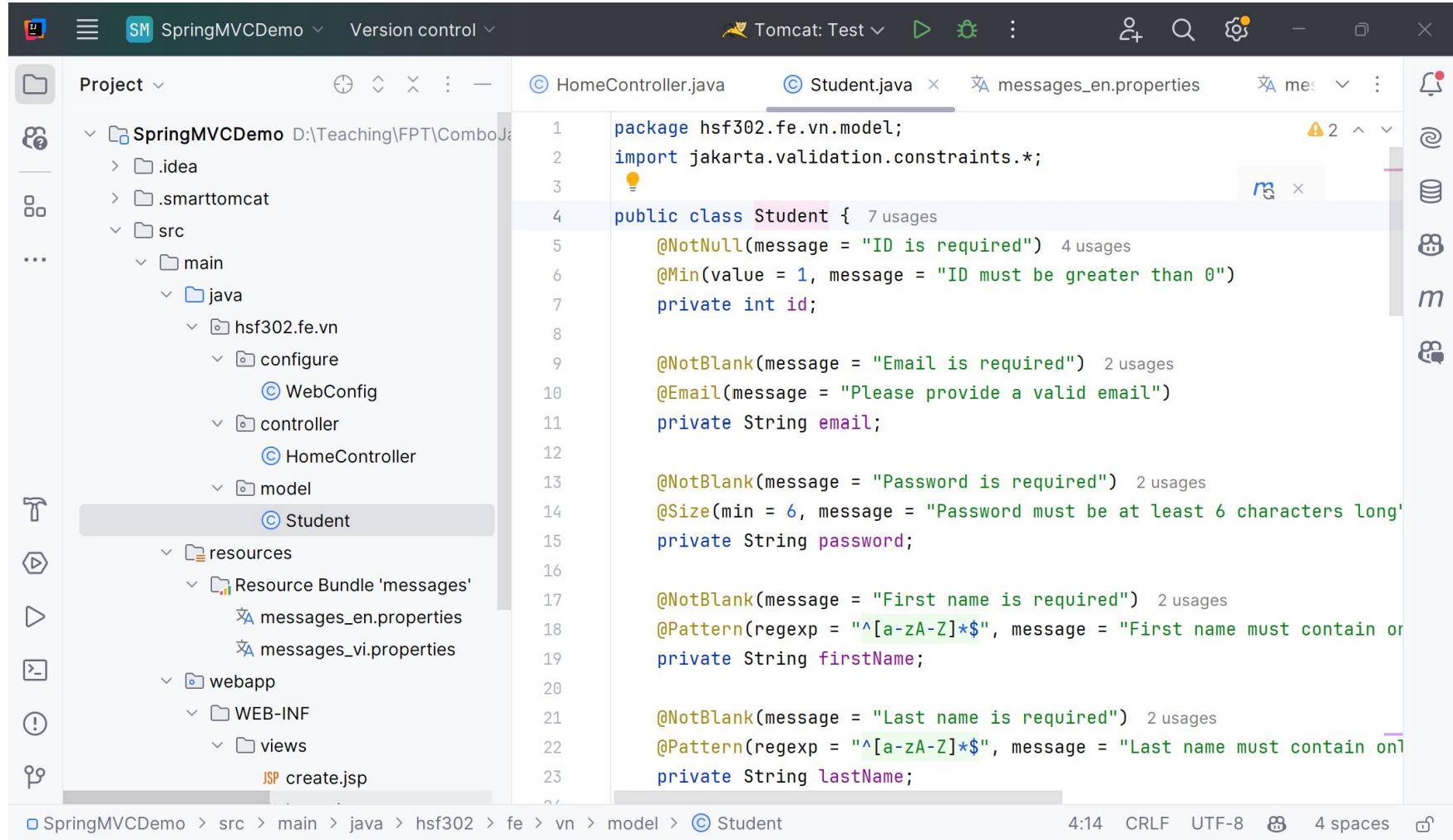
The screenshot shows a Java IDE interface with the following details:

- Project:** SpringMVC Demo
- Editor:** Tomcat: Test
- File:** web.xml
- Content:** XML configuration for a Spring MVC Application.

```
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
          xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
          version="6.0">
    <display-name>Spring MVC Application</display-name>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>hsf302.fe.vn.configure.WebConfig</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

The left sidebar shows the project structure with files like HomeController.java, Student.java, pom.xml, and various resource files. The web.xml file is selected in the project tree.

Create the Student.java



The screenshot shows the IntelliJ IDEA interface with the project 'SpringMVC Demo' open. The 'Student.java' file is selected in the editor tab bar. The code defines a class 'Student' with various fields annotated with validation constraints from the 'jakarta.validation.constraints' package. The code includes annotations for @NotNull, @Min, @NotBlank, @Email, @NotBlank, @Size, @Pattern, and @NotBlank. The 'messages_en.properties' file is also visible in the resources directory.

```
package hsf302.fe.vn.model;
import jakarta.validation.constraints.*;
public class Student {
    @NotNull(message = "ID is required") 4 usages
    @Min(value = 1, message = "ID must be greater than 0")
    private int id;

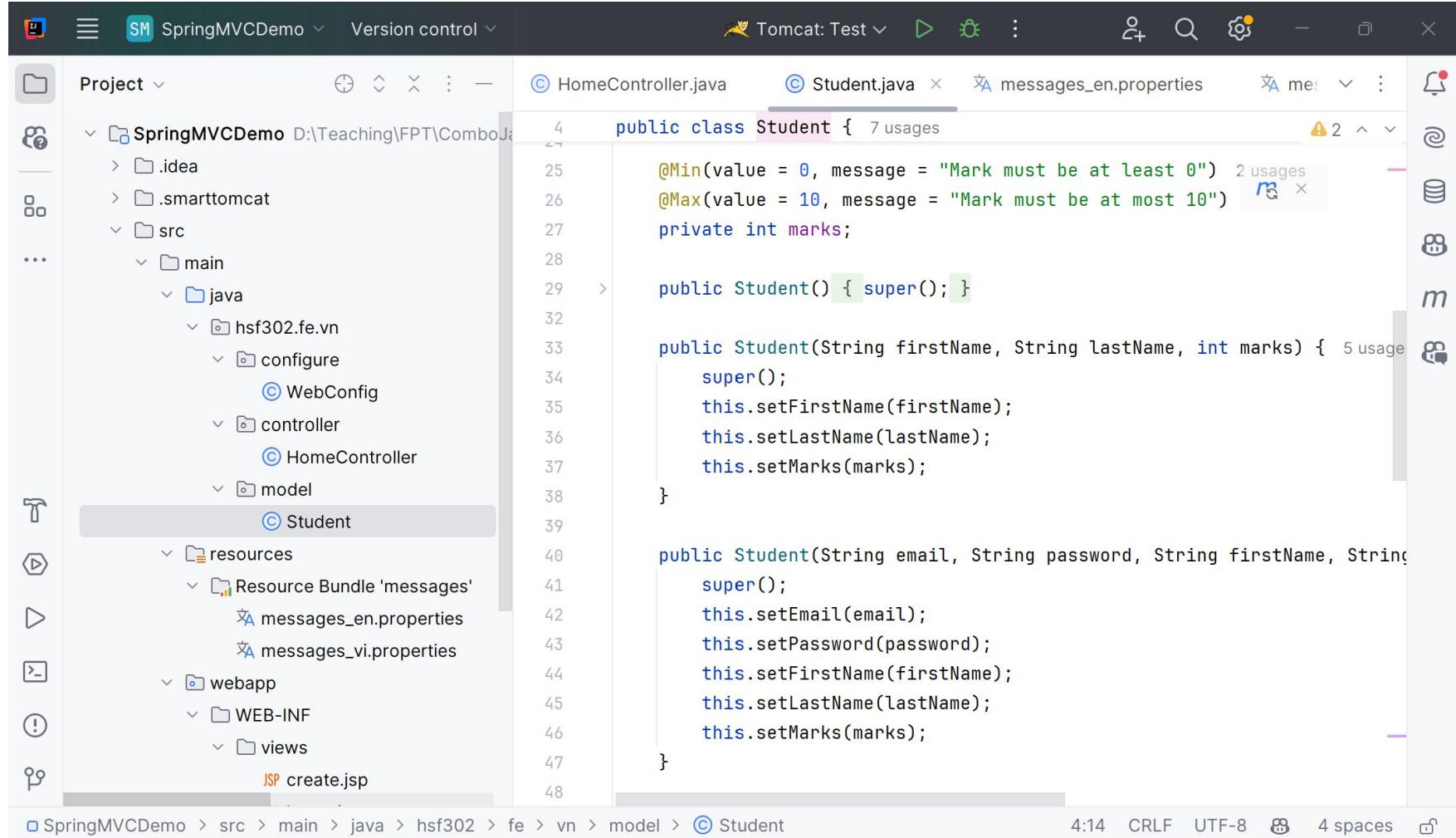
    @NotBlank(message = "Email is required") 2 usages
    @Email(message = "Please provide a valid email")
    private String email;

    @NotBlank(message = "Password is required") 2 usages
    @Size(min = 6, message = "Password must be at least 6 characters long")
    private String password;

    @NotBlank(message = "First name is required") 2 usages
    @Pattern(regexp = "^[a-zA-Z]*$", message = "First name must contain only letters")
    private String firstName;

    @NotBlank(message = "Last name is required") 2 usages
    @Pattern(regexp = "^[a-zA-Z]*$", message = "Last name must contain only letters")
    private String lastName;
```

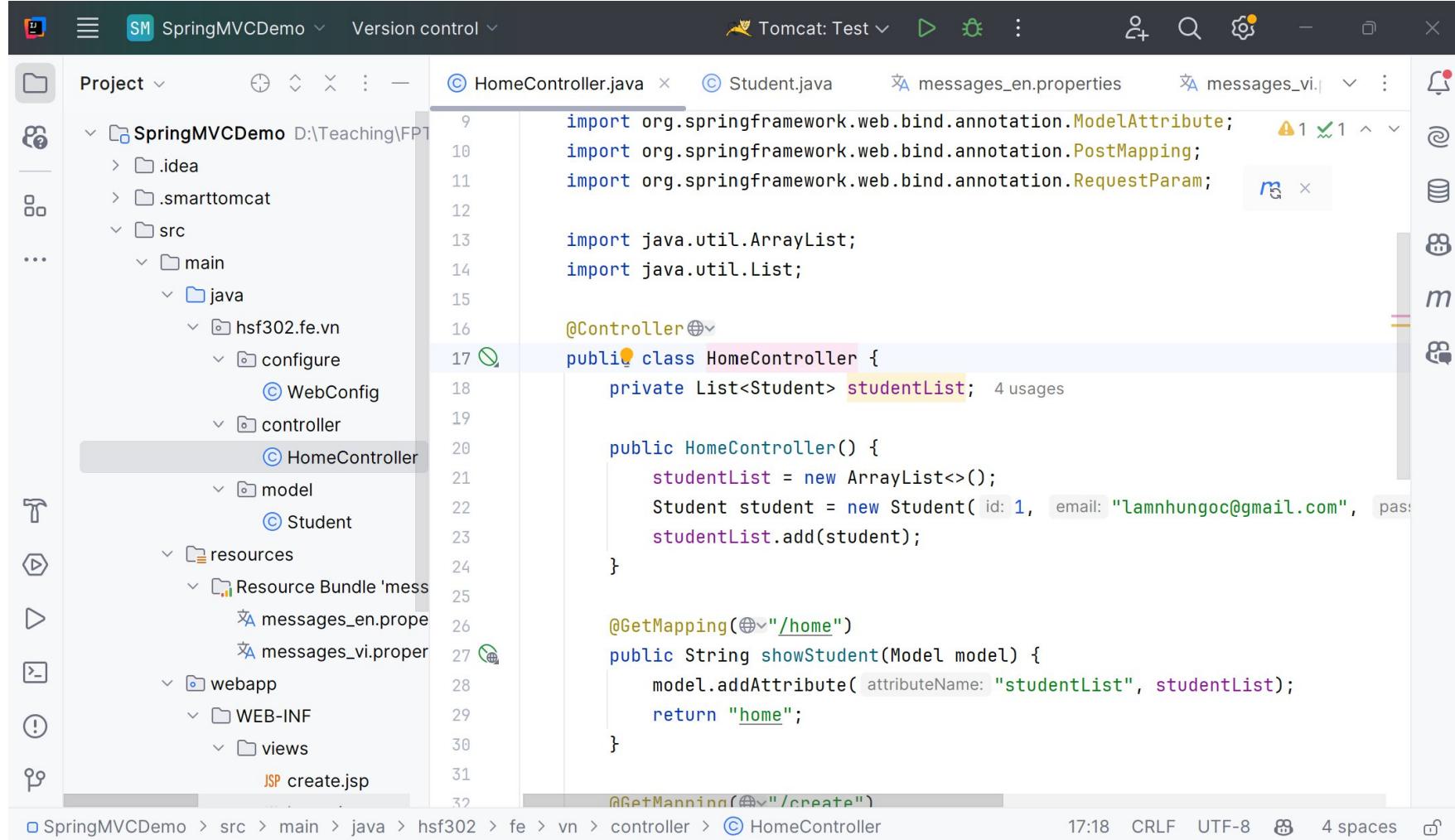
Edit the Student.java



The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Bar:** Shows the project name "SpringMVC Demo" and its location "D:\Teaching\FPT\ComboJava".
- Toolbars:** Version control, Tomcat, and other standard IDE tools.
- Code Editor:** The file "Student.java" is open. The code defines a class "Student" with three constructors and several instance variables annotated with @Min and @Max constraints. A tooltip for the first constructor is visible.
- File Explorer:** Shows the project structure with packages like ".idea", ".smarttomcat", "src", "main", "java", "hsf302.fe.vn", "configure", "WebConfig", "controller", "HomeController", "model", and "Student".
- Properties:** Resource files "messages_en.properties" and "messages_vi.properties" are listed under the "resources" folder.
- Bottom Status Bar:** Displays the file path "SpringMVC Demo > src > main > java > hsf302 > fe > vn > model > Student", and status information like "4:14", "CRLF", "UTF-8", and "4 spaces".

Create HomeController.java



The screenshot shows an IDE interface with the following details:

- Project:** SpringMVC Demo
- File:** HomeController.java
- Content:** The code defines a HomeController class with a constructor that initializes a studentList ArrayList. It contains two @GetMapping annotations: one for "/home" which returns a String "home", and another for "/create" which returns a Model object with a studentList attribute.

```
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import java.util.ArrayList;
import java.util.List;

@Controller
public class HomeController {
    private List<Student> studentList;

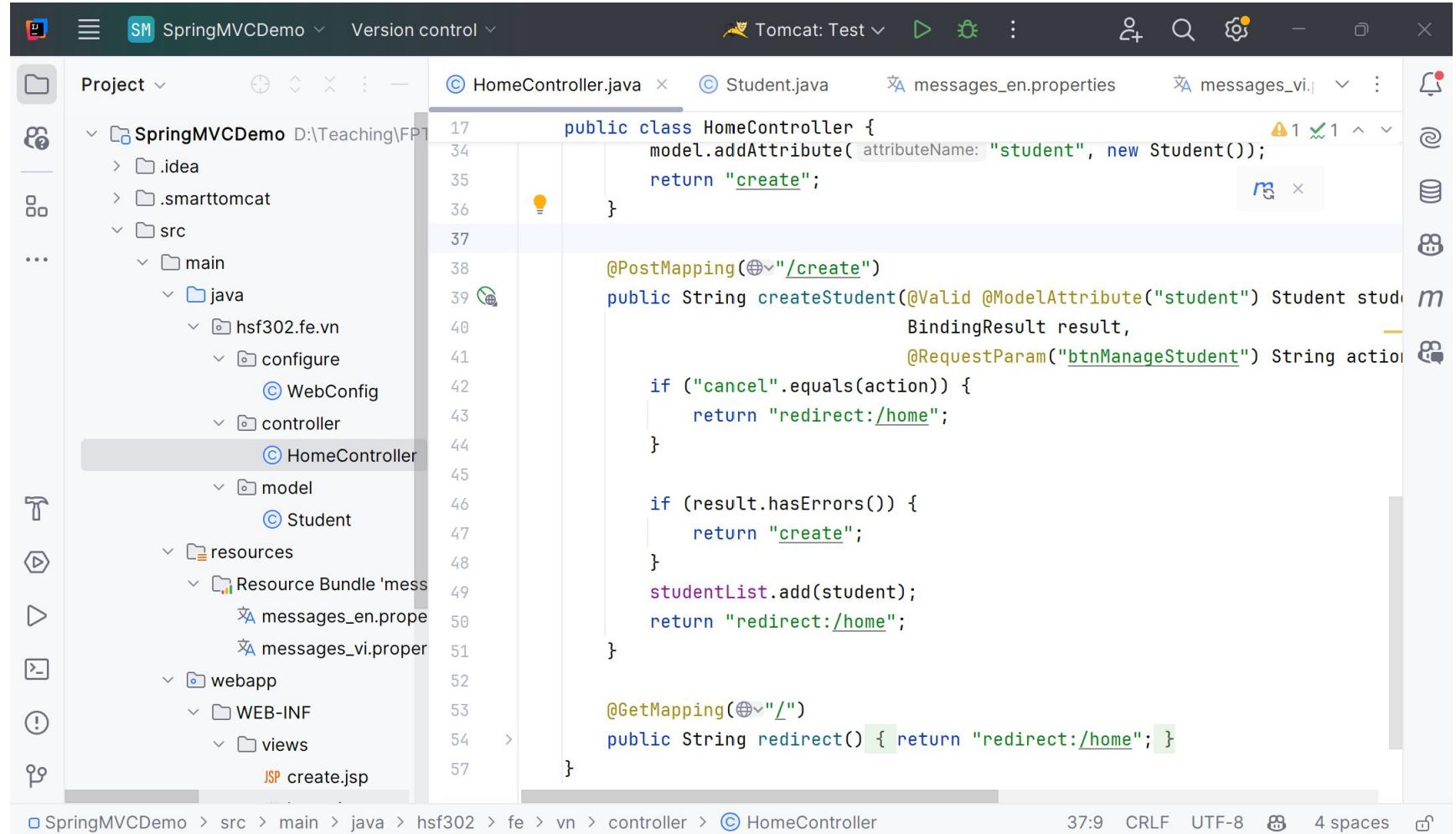
    public HomeController() {
        studentList = new ArrayList<>();
        Student student = new Student( id: 1, email: "lamnhungoc@gmail.com", pass: "123456");
        studentList.add(student);
    }

    @GetMapping("/home")
    public String showStudent(Model model) {
        model.addAttribute( attributeName: "studentList", studentList);
        return "home";
    }

    @GetMapping("/create")
}
```

- IDE Features:** The IDE has a dark theme with various icons for file operations, navigation, and search. A status bar at the bottom shows the file path, time (17:18), encoding (CRLF), and other settings.

Edit HomeController.java



The screenshot shows a Java code editor within an IDE. The project structure on the left includes a .idea folder, a smarttomcat folder, and a src folder containing main, java, resources, and webapp. The java directory under main contains hsf302, fe, vn, controller, model, and Student. The controller directory contains HomeController. The resources directory contains Resource Bundle 'mess' with messages_en.properties and messages_vi.properties. The webapp directory contains WEB-INF and views, with create.jsp in the views folder. The HomeController.java file is open in the editor, showing the following code:

```
public class HomeController {
    model.addAttribute(attributeName: "student", new Student());
    return "create";
}

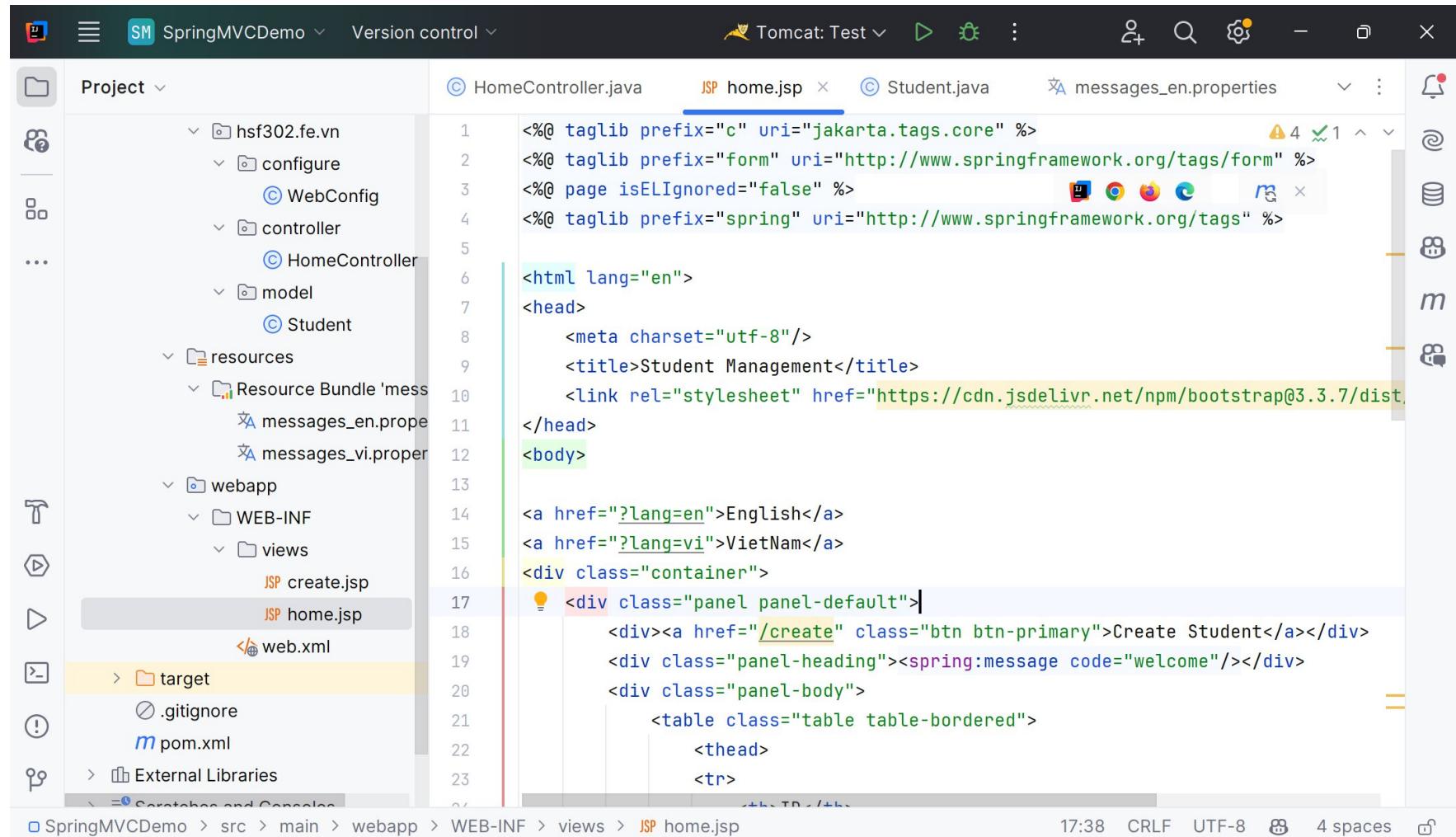
@PostMapping("/create")
public String createStudent(@Valid @ModelAttribute("student") Student student,
                           BindingResult result,
                           @RequestParam("btnManageStudent") String action)
{
    if ("cancel".equals(action)) {
        return "redirect:/home";
    }

    if (result.hasErrors()) {
        return "create";
    }
    studentList.add(student);
    return "redirect:/home";
}

@GetMapping("/")
public String redirect() { return "redirect:/home"; }
```

The code implements a POST mapping for "/create" to handle student creation. It uses a ModelAttribute annotation to bind the "student" object to the view. It also handles a cancel button by returning a redirect to the home page. If there are validation errors, it returns the "create" view again. Otherwise, it adds the student to a list and redirects to the home page. A GET mapping for "/" simply returns a redirect to the home page.

Create home.jsp



The screenshot shows a Java IDE interface with the following details:

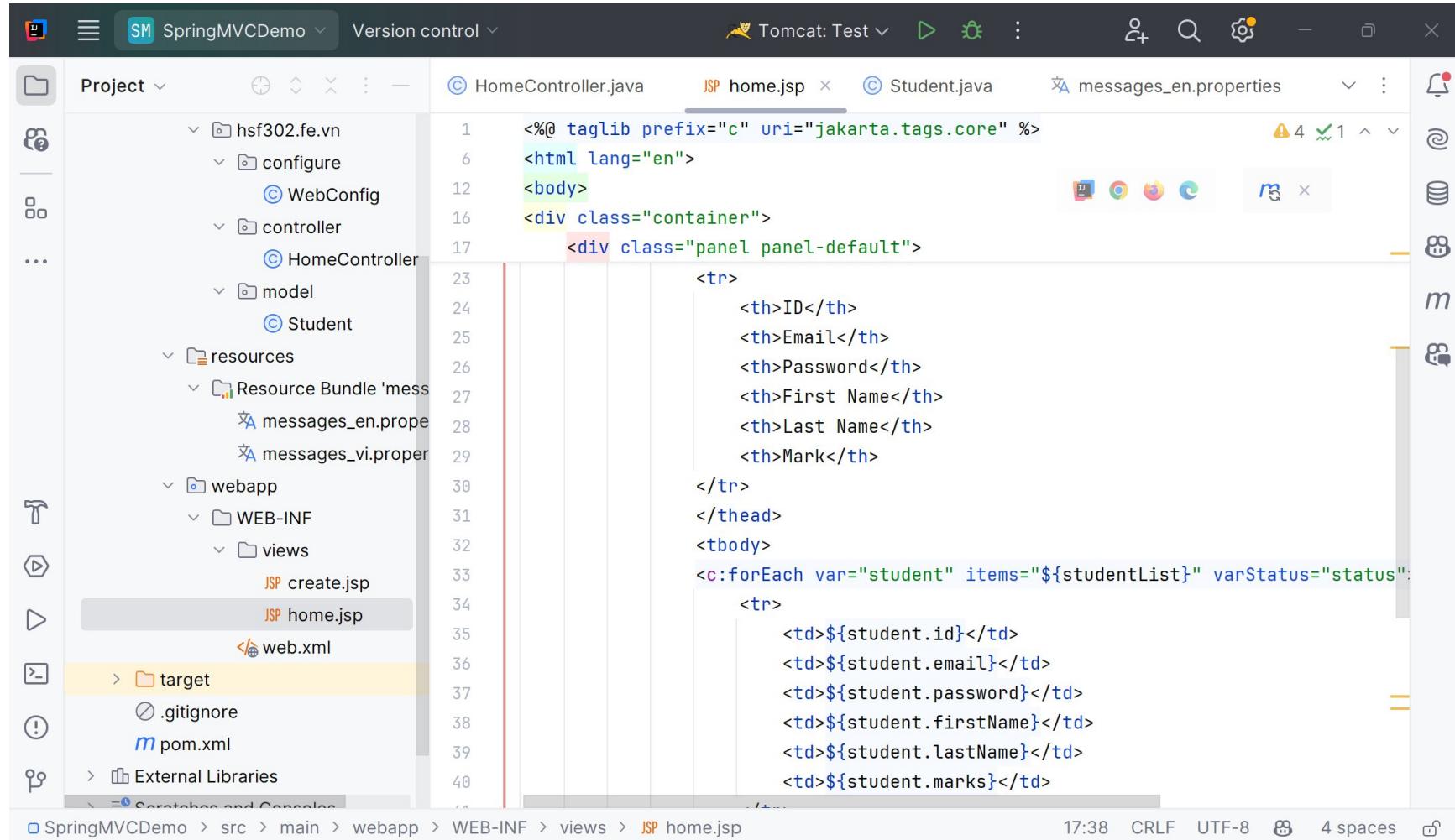
- Project Explorer:** Shows the project structure for "SpringMVC Demo". Key components include "hsf302.fe.vn", "controller" (containing "HomeController"), "model" (containing "Student"), "resources" (containing "Resource Bundle 'mess'"), and "webapp" (containing "WEB-INF" and "views").
- Editor:** The "home.jsp" file is open in the editor. The code is a JSP page with Spring and Bootstrap imports.
- Toolbars and Status Bar:** The top bar includes tabs for "HomeController.java", "JSP home.jsp", "Student.java", and "messages_en.properties". The status bar at the bottom shows the file path "SpringMVC Demo > src > main > webapp > WEB-INF > views > JSP home.jsp", the time "17:38", and encoding "UTF-8".

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>

<html lang="en">
<head>
    <meta charset="utf-8"/>
    <title>Student Management</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css"/>
</head>
<body>

<a href="?lang=en">English</a>
<a href="?lang=vi">VietNam</a>
<div class="container">
    <div class="panel panel-default">
        <div><a href="/create" class="btn btn-primary">Create Student</a></div>
        <div class="panel-heading"><spring:message code="welcome"/></div>
        <div class="panel-body">
            <table class="table table-bordered">
                <thead>
                    <tr>
```

Edit home.jsp

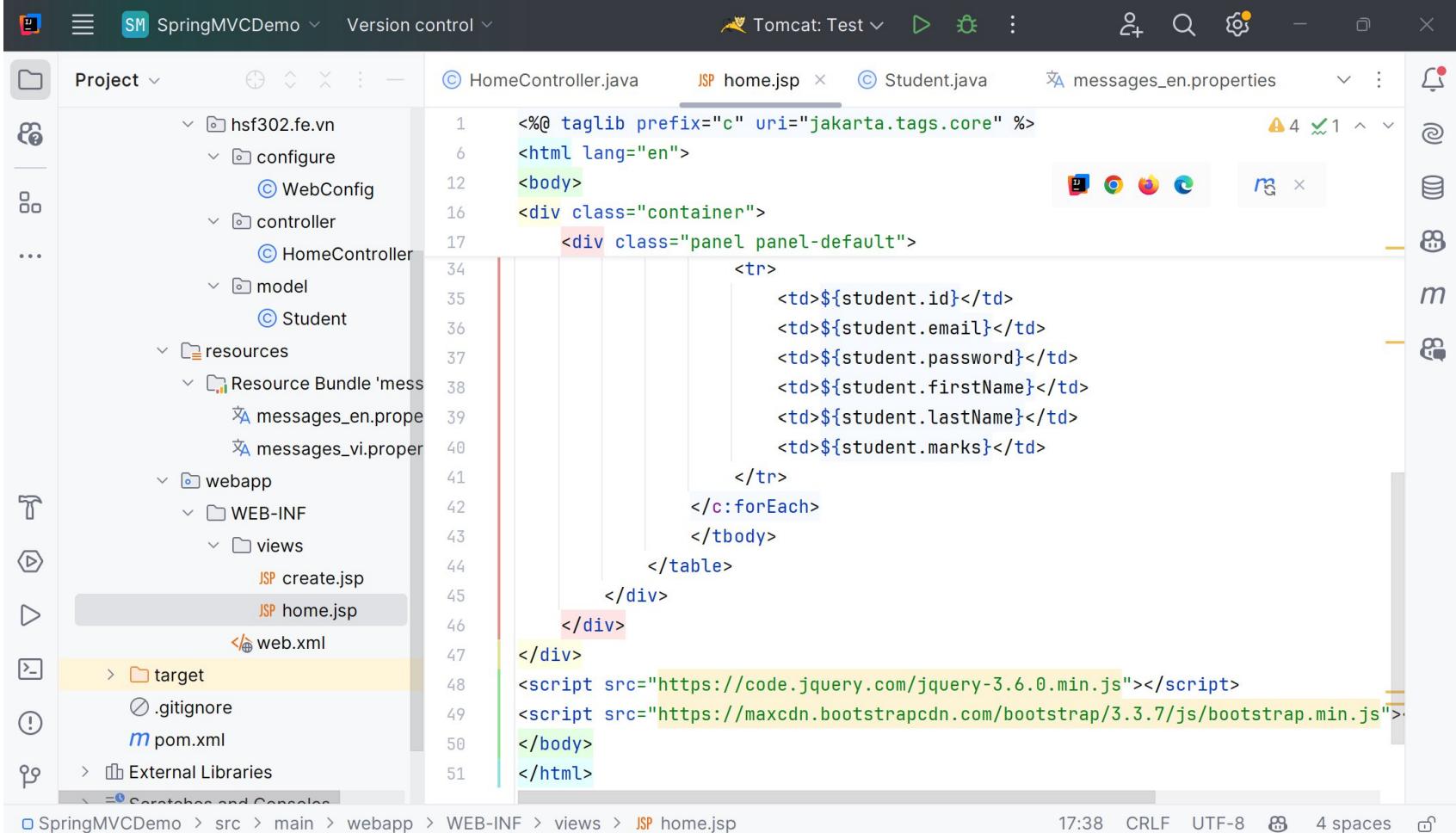


The screenshot shows a Java IDE interface with the following details:

- Project Explorer:** Shows the project structure under "SpringMVC Demo". Key files include `HomeController.java`, `Student.java`, `messages_en.properties`, and several XML configuration files like `WebConfig`, `HomeController`, `Student`, and `web.xml`.
- Code Editor:** The current file is `home.jsp`. The code is a JSP page that includes a JSTL taglib and displays a table of student data.
- Code:**

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html lang="en">
<body>
<div class="container">
<div class="panel panel-default">
<tbl_struct version="1">
<tbody>
<tr>
<th>ID</th>
<th>Email</th>
<th>Password</th>
<th>First Name</th>
<th>Last Name</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<c:forEach var="student" items="${studentList}" varStatus="status">
<tr>
<td>${student.id}</td>
<td>${student.email}</td>
<td>${student.password}</td>
<td>${student.firstName}</td>
<td>${student.lastName}</td>
<td>${student.marks}</td>
</tr>
</c:forEach>
</tbody>
</tbl_struct>
</div>
</div>
</body>
</html>
```
- Status Bar:** Shows the file path as `SpringMVC Demo > src > main > webapp > WEB-INF > views > JSP home.jsp`, and the status bar indicates the file was last modified at 17:38, is in CRLF format, is UTF-8 encoded, has 4 spaces, and includes icons for file operations.

Edit home.jsp



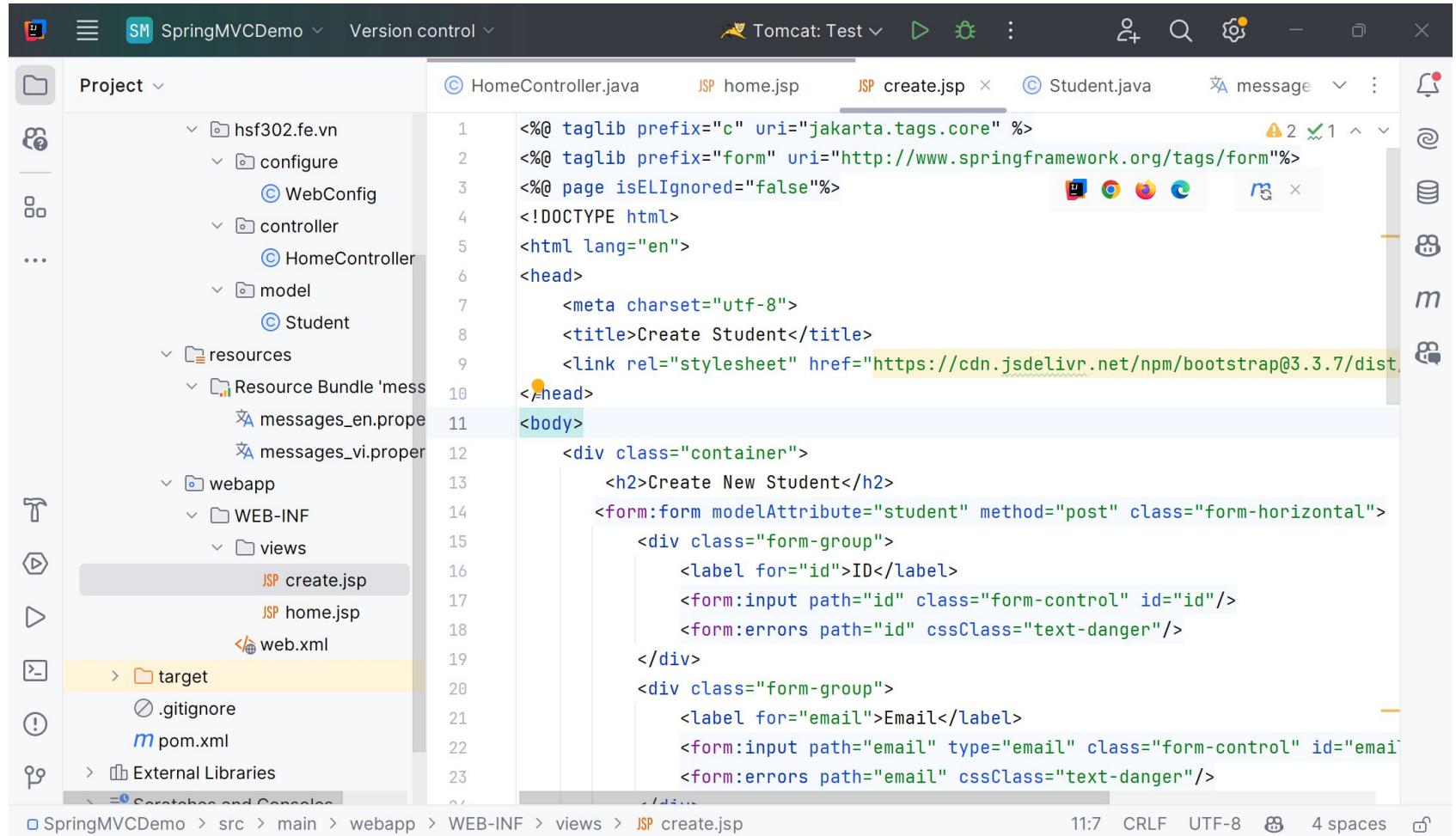
The screenshot shows a Java IDE interface with the following details:

- Project:** SpringMVCDemo
- File:** HomeController.java, Student.java, messages_en.properties
- Selected File:** home.jsp
- Content:** The code is a JSP page that includes a scriptlet, HTML, and JSTL tags. It displays student information from a database using a forEach loop.

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html lang="en">
<body>
<div class="container">
<div class="panel panel-default">
<table border="1">
<tr>
<td>${student.id}</td>
<td>${student.email}</td>
<td>${student.password}</td>
<td>${student.firstName}</td>
<td>${student.lastName}</td>
<td>${student.marks}</td>
</tr>
</c:forEach>
</tbody>
</table>
</div>
</div>
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
</body>
</html>
```

- Toolbars and Status Bar:** The status bar at the bottom shows the path as "SpringMVCDemo > src > main > webapp > WEB-INF > views > JSP home.jsp", and the time as 17:38.

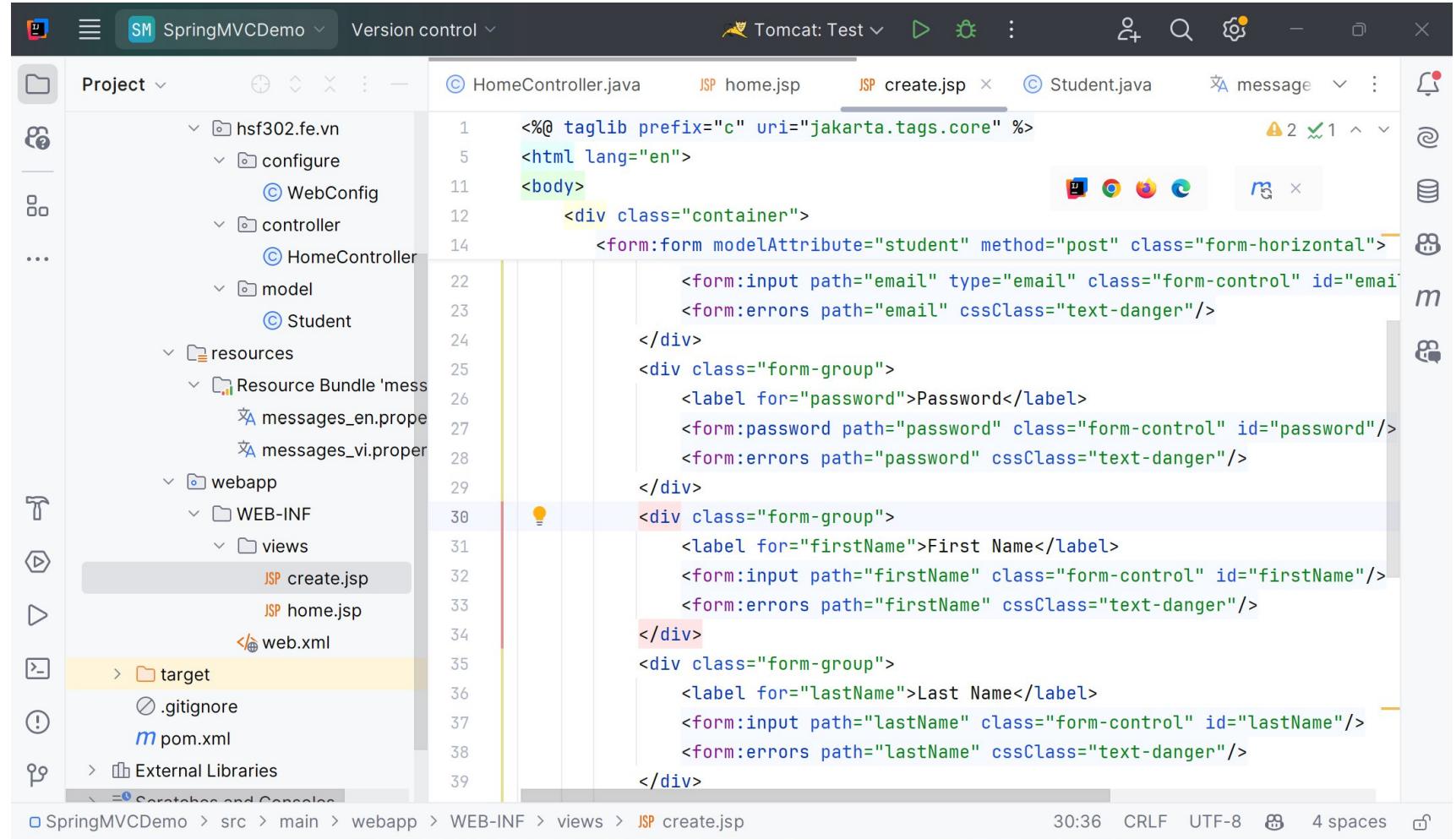
Create create.jsp



The screenshot shows a Java IDE interface with the following details:

- Project Explorer:** Shows the project structure under "SpringMVC Demo". Key files include `HomeController.java`, `home.jsp`, `create.jsp` (which is currently selected), and `Student.java`.
- Code Editor:** Displays the content of the `create.jsp` file. The code is a Spring MVC JSP page for creating a student. It includes imports for taglibs, a DOCTYPE declaration, and a form for inputting student ID and email.
- Toolbar:** Includes standard icons for file operations, search, and refresh.
- Status Bar:** Shows the current file path as `SpringMVC Demo > src > main > webapp > WEB-INF > views > JSP create.jsp`, and system information like "11:7 CRLF UTF-8 4 spaces".

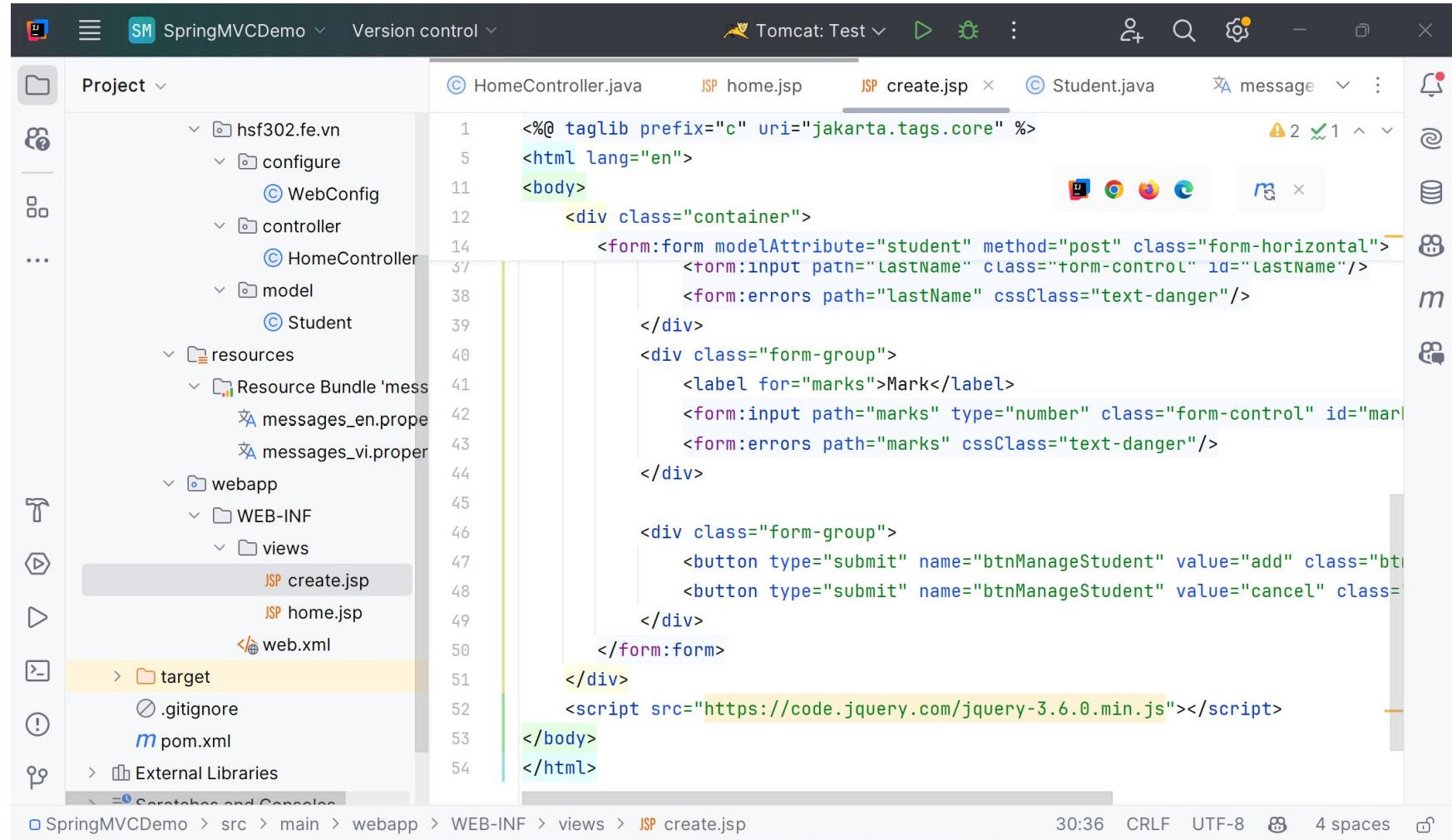
Edit create.jsp



The screenshot shows a Java IDE interface with the following details:

- Project Explorer:** Shows the project structure under "SpringMVC Demo". Key files include `HomeController.java`, `home.jsp`, `create.jsp` (which is the active tab), and `Student.java`. The `create.jsp` file is highlighted.
- Code Editor:** Displays the content of `create.jsp`. The code uses JSP syntax with Jakarta Taglibs, including `<%@ taglib prefix="c" uri="jakarta.tags.core" %>` and various `<form:>` and `<form:input>` tags for form handling.
- Status Bar:** At the bottom, it shows the path `SpringMVC Demo > src > main > webapp > WEB-INF > views > JSP create.jsp`, the time `30:36`, and encoding `CRLF UTF-8`.

Edit create.jsp



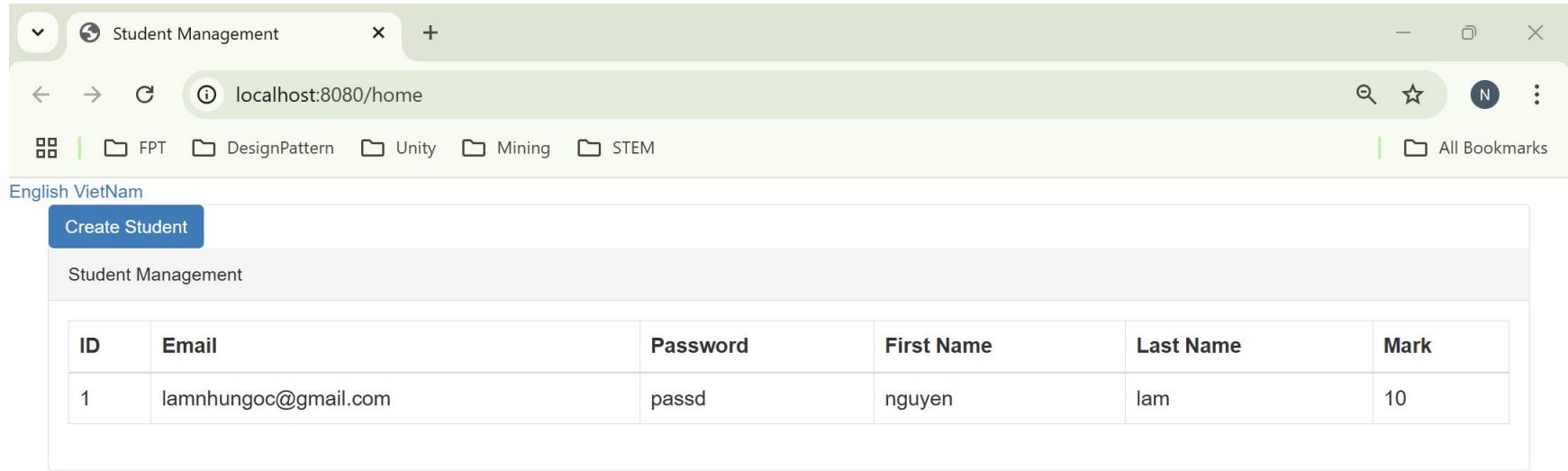
The screenshot shows an IDE interface with the following details:

- Project:** SpringMVC Demo
- File:** create.jsp
- Content:** JSP code using Spring MVC annotations. It includes form fields for 'lastName' and 'marks', and two submit buttons for 'add' and 'cancel'.
- Project Structure:** Shows a tree view with packages like hsf302.fe.vn, controller, model, and Student, along with resources and web.xml.
- Toolbars and Status Bar:** Standard IDE toolbars and status bar showing file path, time (30:36), encoding (CRLF), and other settings.

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html lang="en">
<body>
    <div class="container">
        <form:form modelAttribute="student" method="post" class="form-horizontal">
            <form:input path="lastName" class="form-control" id="lastName"/>
            <form:errors path="lastName" cssClass="text-danger"/>
        </div>
        <div class="form-group">
            <label for="marks">Mark</label>
            <form:input path="marks" type="number" class="form-control" id="marks"/>
            <form:errors path="marks" cssClass="text-danger"/>
        </div>

        <div class="form-group">
            <button type="submit" name="btnManageStudent" value="add" class="btn btn-primary">Add</button>
            <button type="submit" name="btnManageStudent" value="cancel" class="btn btn-primary">Cancel</button>
        </div>
    </form:form>
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
</body>
</html>
```

Run Program



Student Management

localhost:8080/home

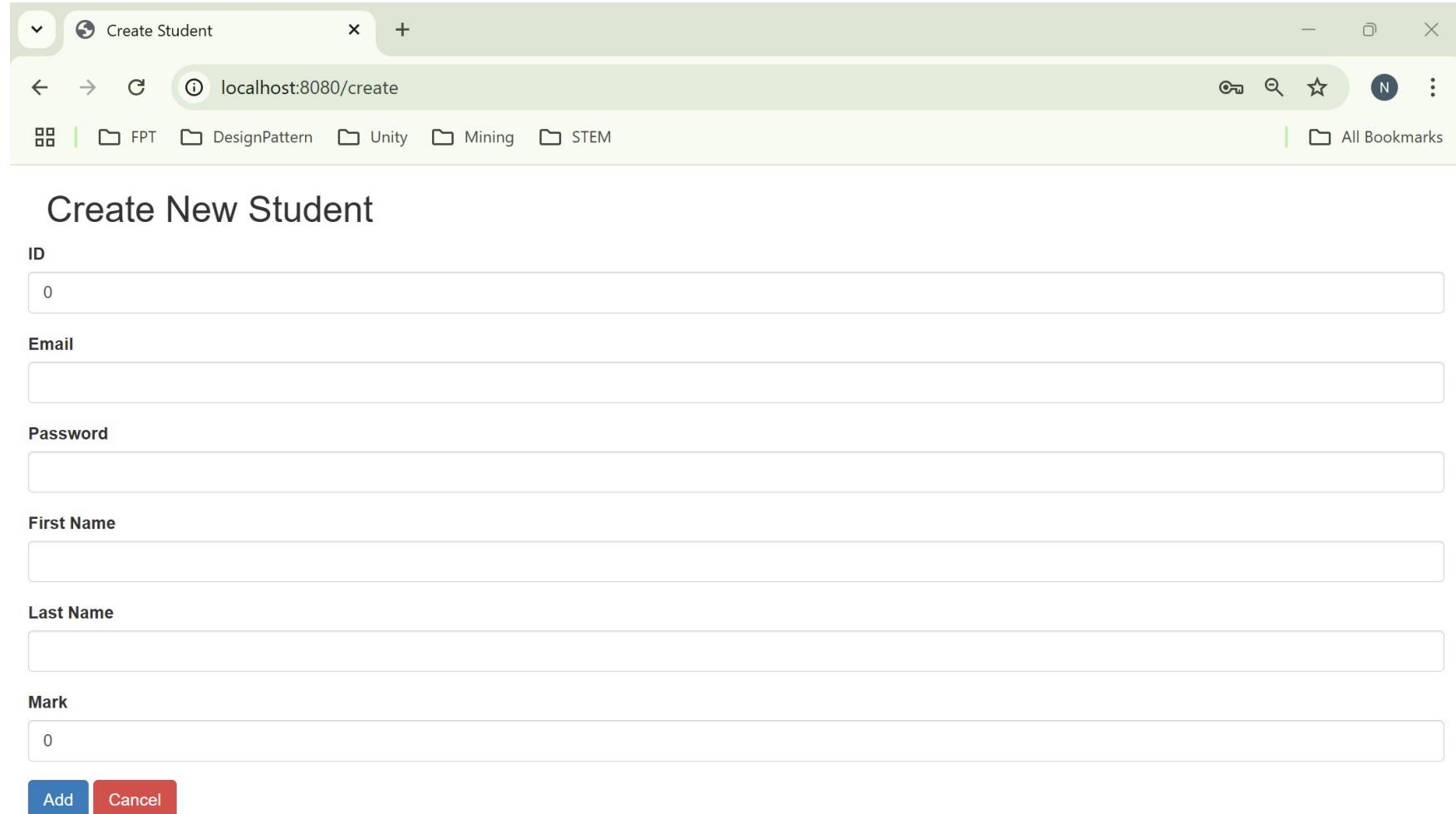
FPT DesignPattern Unity Mining STEM All Bookmarks

Create Student

Student Management

ID	Email	Password	First Name	Last Name	Mark
1	lamnhungoc@gmail.com	passd	nguyen	lam	10

Run Program



The screenshot shows a web browser window titled "Create Student" with the URL "localhost:8080/create". The browser interface includes a back/forward button, a search bar, and a bookmarks bar with links to "FPT", "DesignPattern", "Unity", "Mining", and "STEM". Below the header, the main content area displays a "Create New Student" form with the following fields:

- ID**: An input field containing the value "0".
- Email**: An empty input field.
- Password**: An empty input field.
- First Name**: An empty input field.
- Last Name**: An empty input field.
- Mark**: An input field containing the value "0".

At the bottom of the form are two buttons: a blue "Add" button and a red "Cancel" button.

Summary

Concepts were introduced:

- ◆ Spring MVC Basics
- ◆ Spring MVC Framework
- ◆ Controller/Model/View
- ◆ Spring Interceptor
- ◆ Spring Validator