Model-View-Controller design pattern

* design pattern? It is proven solution for a recursive problem.
* design pattern is not an API/technology/Framework. It is just a set of principles to solve a recursive problem.
* design patterns are divided 4 categories.
1. creational design patterns
2. structural design patterns
3. behavioral design patterns
4. Java EE patterns.
* The MVC is a design pattern for building the user interfaces.
* The MVC pattern segregates an application into interconnected components, Model, View and Controller, to achieve modularity and to make the application more flexible and scalable.

Model: represents the data and the business logic of the application.

* The classes in a project, which are holding the data or enforces the business rules or interacting with the database, are Model classes.
* For example, in a Banking system, the classes we create like Account or Transaction or BankService, etc.. are the Model classes.

View:

* The View is responsible for displaying the information for the user.
* The View represents the presentation layer of the application.
* The View gets the data from the Model and will present it to the user.
* The View could be a web page that shows the data to the user, or it could be a web page that shows a form to accept the user’s input.
* For example, a web page showing the account details is a View, or a web page showing the form for money transfer is a View.

Controller:

* The Controller acts as a mediator between the Model and the View.
* It will process the user input, it will interact with the Model and it will update the View.
* It manages the entire request flow of an application.

For example, The Controller handles the requests for viewing account balance, or making money transfers, etc.



Spring MVC flow:

* The components in Spring MVC flow are,
1. DispatcherServlet
2. HandlerMapping
3. Controller
4. ViewResolver
5. View





* A user sends an HTTP request, by entering the URL in the browser(Client).
* The request first lands at the DispatcherServlet.
* DispatcherServlet is a Front Controller in Spring MVC, and it is C in MVC.
* DispatcherServlet is responsible for directing the request to the right Controller class(this controler is M in MVC), who can handle the request.
* The DispatcherServlet class consults with the HandlerMapping component, to find the right controller class.
* Now the DispatcherServlet forwards the request to that controller class.
* The controller class, may directly contain business logic and interacts with the database to fetch data or it can invoke a service class, then intern repository class, to fetch the data from the database.
* The controller class returns, the view name or the view name and the data to the DispatcherServlet.
* The DispatcherServlet, now consults with the ViewResolver to determine the view(JSP, Thymeleaf, freemarker,etc.).
* The DispatcherServlet, now includes the reponsse of a view. Finally sends that reponse to the client.

How to create a controller class:

* @Controller
* @RequestMapping

@Controller is used at class-level and @RequestMapping can be used at class-level or at method-level.

The controller class created in spring mvc project, is responsible to process the input and return appropriate view name or the data.

* The controller class can have the business logic or it can interact with service classes to handle the request.
* @Controller is a stereo type annotation, it marks the a class as a controller in a spring mvc project.
* @RequestMapping maps a request to a specific controller method based on the URL and the HTTP method.

for example:

@RequestMapping(value=”/hello”, method = RequestMethod.GET) 🡪 It maps a url /hello with HTTP method GET on to a controller method.

@Controller

public class HelloController {

 @RequestMapping(“/hello”)

 public String sayHello() {

 return “welcome”;

 }

}



https://github.com/ShekherJava/SB-WelcomeMVC.git

From spring 4.2, we can also use short cut annotations for @RequestMapping.

@RequestMapping(value = “/hello”, method = RequestMethod.GET) ---> @GetMapping(“/hello”)

@RequestMapping(value = “/hello”, method = RequestMethod.POST) ---> @PostMapping(“/hello”)

@RequestMapping(value = “/hello”, method = RequestMethod.PUT) ---> @PutMapping(“/hello”)

@RequestMapping(value = “/hello”, method = RequestMethod.DELETE) ---> @DeleteMapping(“/hello”)

@RequestMapping(value = “/hello”, method = RequestMethod.PATCH) ---> @PatchMapping(“/hello”)



<https://github.com/ShekherJava/SB-LoginMVC.git>

=======================================



https://github.com/ShekherJava/SB-LoginMVC-Database.git

Bean validation api:

* client-side validation
* server-side validation
* client-side validation can be implemented in javascript, it is very useful and faster.
* Relying entirely on client-side validations makes our application vulnerable. Because, users can disable javascript on their browser or the validation logic can be tampered on the client-side.
* For security, data integrity, consistency, an application required both client-side and server-side validations.
* In spring, the server-side validations can be implemented using Bean validation api.
* Bean validation api, is a set of annotations applied on the Java bean properties/methods, to implement the constraints.

@NotNull: The value of a field or a property must not be null.

 public class User {

 @NotNull

 private String firstName;

 @NotNull

 private String lastName;

}

@Size : the field or property should contain the specified minimum and maximum length.

public class User {

 @Size(min=2, max=12)

 private String username;

}

@Min / @Max: The integer field should contain its value >= min value or <= max value.

public class User {

 @Min(18)

 @Max(28)

 private int age;

}

@Email: The value of a field/property should be a valid email address.

public class User {

 @Email

 private String email;

}

@Past: The value of the field/property should be a data in the past.

public class User {

 @Past

 private LocalDate dateOfBirth;

}

@AssertTrue: The value of a boolean field must be true.

public class User {

 @AssertTrue

 private boolean isActive;

}

@DecimalMin/@DecimalMax: The value of a decimal field should be >=min and <=max value.

public class Product {

 @DecimalMin(“0.01”)

 @DecimalMax(“999.99”)

 private double price;

}

@Pattern: The value of the string field, must match with the specified regular expression.

public class User {

 @Pattern(regexp = “^[A-Z][a-z0-9]\*”)

 @Size(min=6)

 private String password;

 }

@NotEmpty: The value of a string field should not be empty.

public class User {

 @NotEmpty

 prviate String username;

 @NotEmpty

 private String password;

}

Data binding:

* Data binding is a feature in spring MVC, and it refers to the automatic process of converting data from the request into Java object.
* It binds the form data to the model object directly without manually parsing the request.
* When a user submits a form, the data of the form fields will be automatically bound to the model object.
* To automatically bind the form and the Java object, the names of the form fields and the properties of the Java object should match.
* @ModelAttribute of Spring MVC, binds the form data to a model object and makes it available to the controller method, when a form is submitted.

for ex:

 @PostMapping(“/authenticate”)

 public String authenticateUser(@ModelAttribute User user) {

 //logic

 }

* Spring has provided a set of form tags with in a tag library called Spring Form Tag library.
* These tags will simplify the bindings between the form fields and the model attributes.(The variables in the Java class/Model class).
* To use spring form tags, we need to import the namespace of spring form tag library into your jsp file.

<%@ taglib prefix=”form” uri=”http://www.springframework.org/tags/form”%>

<form:form action=”authenticate” modelAttribute=”user”>

 Username: <form:input path=”username”/> <br>

 Password: <form:password path=”password”/><br>

 <input type=”submit” value=”submit”>

</form:form>

Defining the error messages:

* 1. within validation annotation
* 2. within messages.properties file

public class User {

 @NotEmpty(message=”Username is required”)

 private String username;

 @Size(min=2, max=8, message=”Password should contain atleast 2 and atmost 8 characters”

 private String password;

 //setters and getters

}

* The issue here is, if you would like to change the message then you have to the modifications in the source code. So, the recommendation is, define the validation error messages in a properties file.
* In the messages.properties file, the key of a validation error message should follow a syntax.

 constraint.modelAttribute.property=message

for ex:

NotEmpty.user.username=Username is required

Size.user.password=Password should contain atleast 2 and atmost 8 characters

==========================================



<https://github.com/ShekherJava/SB-Validation.git>

Spring Boot Dev Tools:

* This module is designed to speed up the development by automatically restarting the application, whenever the files on the classpath are changed.
* If we make any changes to the project files, after application started, then we have to manually restart the full application.
* This manual restart may take some time, if application is too large.
* So, we got a solution as Spring Boot Dev tools module.
* If we add this dependency, then Spring Boot detects the changes in the project files, and reloads only the changed parts, not the entire application. So, the application restart is faster.
* This module is intended for development use, not for production. So, if we package an application into .jar/.war file, then Spring Boot will no load this module into that .jar/.war file.

<dependency>

 <groupId>org.springframework.boot</groupId>

 <artifactId>spring-boot-devtools</artifactId>

 <optional>true</optional>

</dependency>

* The <optional>true</optional> tag is used to signal that this dependency will not be inherited to the othe project, when the other project is using this project as a dependency.
* If you don’t specify <optional>true</optional> tag, then by default the value is false. It means, the spring-boot-devtools dependency will be automatically inherted to other projects, when they use this project as a dependency.

File uploading/downloading using Database:

* The key things to do:

 Database setup, like creating a table to store the file metadata.

 Add the dependencies like web, Data JPA, MySQL Driver, JSP, JSTL, Dev tools, Lombok.

 Create Entity, Repository, Serivce and a Controller class to handle file uploading and file downloading.

 Create the JSP files, to display the upload form and to list the uploaded files for download.

use test;

CREATE TABLE FILES (

ID INT auto\_increment,

FILE\_NAME VARCHAR(100),

FILE\_TYPE VARCHAR(100),

FILE\_DATA BLOB,

PRIMARY KEY(ID));

* In HTML form desing, we have to create a file field to choose the file and upload it to the server.
* <input type=”file” name=”file”>
* The HTTP method “post” is used to upload a file.
* The HTML form should use encoding type

“multipart/form-data” to allow file uploads.

* multipart/form-data allows files, images and other binary data to be transmitted along with other text data to the server.
* multipart means, the form data is divided into mulitple parts. Each part contains form field and its corresponding value.
* if a file is included, one of the parts contain the file data.
* multipart/form-data ensures that both textual data and the binary file data can be sent in the same HTTP request.
* When a file is uploaded in a request, Spring recevies the uploaded file in multipart request into MultipartFile object.
* The MultipartFile object contains, the actual data of the uploaded file, and meta information like, original file name and content type, and the size of the uploaded file in bytes.



<https://github.com/ShekherJava/SB-Fileupload.git>