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;

}