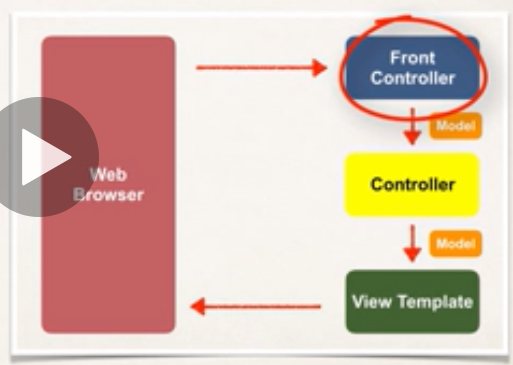
**Spring MVC**



* Front Controller is also known as “**Dispatcher Servlet**”
* It is the part of Spring Framework
* **Controller**
* Controller handles Web request
* Contains Business logic (Service and Dao layer)
* Dao layer save/fetch data from DB... place the data back in **Model**
* Send back the view component to web browser
* **Spring MVC- Config**
* **Web.xml**
* **Step 1: Configure Spring MVC Dispatcher Servlet**
* **Step 2: Set up URL mapping for Spring MVC Dispatcher Servlet**
* **Xx-servlet.xml**
* **Step 3: Add support for component scanning**
* **Step 4: Add support for conversion, formatting and validation support**
* **Step 5: Define Spring MVC view resolver**

**Spring – Validation**

1. **In-build Validation**

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| **@NotNull** | **Check that the annotated value is not null** |
| **@Max** | **Number >= value** |
| **@Min** | **Number < = value** |
| **@Size** | **Size must match the given size** |
| **@Pattern** | **Must match the regular expression pattern** |
| **@Future/@Past** | **Date must be in future or past the given date** |

1. **Development process for in-build validation**

**Step 1:- Add Validation rule to Model Class**

**Ex-**

Public class Student{

private String firstName;

//where last name is the required field means that should not be null

@NotNull(message=”is required”) // error message if validations fails

@Size(min=1, message=”is require”)

Private String lastName;

}

Step 2:- Display error message on HTML form

**Step 3:- Perform validation on controller class**

@RequestMapping(“/students)

Public void sampleRequest(**@Valid** @ModelAttribute(“student”) Student theStudent , **@BindingResult** theBindingResult){

If(theBindingResult.hasError()){

Return “student-error page”;

}

Else{

Return “student-confirmation page”;

}

}

-> **@Valid** = perform validation rule on Student object

-> **@BindingResul**t = Results of validation placed in binding result

Step 4: Update Confirmation jsp/view page

* **Custom Annotation**

**Step 1:- Create an annotation**

**@Constraint(validatedBy = CourseCodeConstraintValidator.class)**

**@Target( { ElementType.METHOD, ElementType.FIELD } )**

**@Retention(RetentionPolicy.RUNTIME**)

public @interface CourseCode {

// define default course code

public String value() default "LUV";

// define default error message

public String message() default "must start with LUV";

// define default groups

public Class<?>[] groups() default {};

// define default payloads

public Class<? extends Payload>[] payload() default {};

}

**Step 2 :** - Create a Class that implements the interface **ConstraintValidator**

import javax.validation.ConstraintValidator;

import javax.validation.ConstraintValidatorContext;

**public class CourseCodeConstraintValidator**

**implements ConstraintValidator<CourseCode, String>** {

private String coursePrefix;

@Override

public void initialize(CourseCode theCourseCode) {

coursePrefix = theCourseCode.value(); //theCourceCode.value() = @CourceCode(value = "LUV")

}

@Override

public boolean isValid(String theCode,//theCode is actual uses entry value/input value

ConstraintValidatorContext theConstraintValidatorContext //we can place additional error message here) {

boolean result;

//business logic for validation

if (theCode != null) {

result = theCode.startsWith(coursePrefix);

}

else {

result = true;

}

return result;

}

}

**Spring – Exception Handling**

**1.**

* Step by step process
* **Step 1: Create an error response class**
* public class ErrorResponse {
* private int status;
* private String message;
* private long timeStamp;}
* **Step 2: - Create a custom exception class**
* public class StudentException **extends RuntimeException**{
* public StudentException(String message, Throwable cause) {
* super(message, cause);
* }
* public StudentException(String message) {
* super(message);
* }
* public StudentException(Throwable cause) {
* super(cause);
* }

* }
* **Step 3: Create a Global exception handler class by using @ControlAdvice annotation and @ExceptionHandler**
* @ControllerAdvice
* public class CustomValidationExceptionHandler {
* @ExceptionHandler(StudentException.class)
* public ResponseEntity<ErrorResponse> exceptionHandle(StudentException exception){
* ErrorResponse error = new ErrorResponse();
* error.setStatus(HttpStatus.NOT\_FOUND.value());
* error.setMessage(exception.getMessage());
* error.setTimeStamp(System.currentTimeMillis());
* return new ResponseEntity<>(error , HttpStatus.NOT\_FOUND);
* }
* @ExceptionHandler
* public ResponseEntity<ErrorResponse> handleException(Exception ex){
* ErrorResponse error = new ErrorResponse();
* error.setStatus(HttpStatus.INTERNAL\_SERVER\_ERROR.value());
* error.setMessage(ex.getMessage());
* error.setTimeStamp(System.currentTimeMillis());
* return new ResponseEntity<>(error , HttpStatus.INTERNAL\_SERVER\_ERROR);
* }
* }

2.

* **Custom Validation – Exception Handling**
* **Step 1: Create an error response model class**
* private String message;
* private int status;
* private long timeStamp;
* private List<String> details = new ArrayList<>();
* **Step 2:- Create a custom validation class by extending the class** **ResponseEntityExceptionHandler**
* **@ControllerAdvice**
* public class CustomValidationExceptionHandler extends ResponseEntityExceptionHandler {
* @Override
* protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request) {
* List<String> details = new ArrayList<>();
* for(ObjectError error : ex.getBindingResult().getAllErrors()) {
* details.add(error.getDefaultMessage());
* }
* ErrorResponse error = new ErrorResponse();
* error.setMessage("Validation Failed..Code must be starts with LUV");
* error.setDetails(details);
* return new ResponseEntity<Object>(error, HttpStatus.BAD\_REQUEST);
* }
* }