Thymeleaf with Spring Boot

© luv2code LLC

What is Thymeleaf?

- Thymeleaf is a Java templating engine

www.thymeleaf.org

Separate project
Unrelated to spring.io

- Commonly used to generate the HTML views for web apps

- However, it is a general purpose templating engine
  - Can use Thymeleaf outside of web apps (more on this later)

© luv2code LLC

What is a Thymeleaf template?

- Can be an HTML page with some Thymeleaf expressions
- Include dynamic content from Thymeleaf expressions

© luv2code LLC

Where is the Thymeleaf template processed?

- In a web app, Thymeleaf is processed on the server
- Results included in HTML returned to browser

© luv2code LLC

Thymeleaf Demo

Output

Time on the server is Sun Jan 06 17:00:40

© luv2code LLC

Development Process

Step-By-Step

1. Add Thymeleaf to Maven POM file

2. Develop Spring MVC Controller

3. Create Thymeleaf template

© luv2code LLC

Step 1: Add Thymeleaf to Maven pom file

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Based on this,
Spring Boot will auto configure to use Thymeleaf templates

Dependencies

Spring Web WEB

Build web, including RESTful, applications using Spring MVC.
Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES

A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

© luv2code LLC

Step 2: Develop Spring MVC Controller

File: DemoController.java

```
@Controller
public class DemoController {

  @GetMapping("/")
  public String sayHello(Model theModel) {

    theModel.addAttribute("theDate", new java.util.Date());

    return "helloworld";
  }
}
```

```
src/main/resources/templates/helloworld.html
```


© luv2code LLC

Where to place Thymeleaf template?

- In Spring Boot, your Thymeleaf template files go in
  - **src/main/resources/templates**

- For web apps, Thymeleaf templates have a **.html** extension

© luv2code LLC

Step 3: Create Thymeleaf template

Thymeleaf accesses "theDate" from the Spring MVC Model

File: DemoController.java

```
@Controller
public class DemoController {

   @GetMapping("/")
   public String sayHello(Model theModel) {

      Attribute("theDate", new java.util.Date());

      world";

   }
```

To use Thymeleaf expressions

1

File: src/main/resources/templates/helloworld.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head> … </head>

<body>
    <p th:text="'Time on the server is ' + ${theDate}" />
</body>

</html>
```

Thymeleaf expression

2

Time on the server is Sun Jan 06 17:00:40

© luv2code LLC

Step 3: Create Thymeleaf template

Thymeleaf accesses "theDate" from the Spring MVC Model

File: DemoController.java

```
@Controller
public class DemoController {

   @GetMapping("/")
   public String sayHello(Model theModel) {

      theModel.addAttribute("theDate", new java.util.Date());

      return "helloworld";
   }
}
```

1

File: src/main/resources/templates/helloworld.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head> … </head>

<body>
    <p th:text="'Time on the server is ' + ${theDate}" />
</body>

</html>
```

2

Thymeleaf expression

Time on the server is Sun Jan 06 17:00:40

© luv2code LLC

Additional Features

- Looping and conditionals
- CSS and JavaScript integration
- Template layouts and fragments

www.thymeleaf.org

© luv2code LLC

CSS and Thymeleaf

© luv2code LLC

Let's Apply CSS Styles to our Page

Before

After

```
font-style: italic;
color: green;
```


© luv2code LLC

Using CSS with Thymleaf Templates

- You have the option of using
  - Local CSS files as part of your project
  - Referencing remote CSS files

- We'll cover both options in this video

© luv2code LLC

Development Process

Step-By-Step

1. Create CSS file

2. Reference CSS in Thymeleaf template

3. Apply CSS style

© luv2code LLC

Step 1: Create CSS file

- Spring Boot will look for static resources in the directory
  - **`src/main/resources/static`**

You can create your own
custom sub-directories
`static/css`
`static/images`
`static/js`
etc …

Can be any sub-directory name

File: demo.css

```
.funny {
  font-style: italic;
  color: green;
}
```


© luv2code LLC

Step 2: Reference CSS in Thymeleaf template

File: helloworld.html

```
<head>
  <title>Thymeleaf Demo</title>

  <!-- reference CSS file -->
  <link rel="stylesheet" th:href="@{/css/demo.css}" />

</head>
```

@ symbol
Reference context path of your application (app root)

© luv2code LLC

Step 3: Apply CSS

File: helloworld.html

```
<head>
  <title>Thymeleaf Demo</title>

  <!-- reference CSS file -->
  <link rel="stylesheet" th:href="@{/css/demo.css}" />
</head>

<body>
  <p th:text="'Time on the server is ' + ${theDate}" class="funny" />
</body>
```

File: demo.css

```
.funny {
  font-style: italic;
  color: green;
}
```

© luv2code LLC

Other search directories

Spring Boot will search following directories for static resources:

`/src/main/resources`

1. `/META-INF/resources`
2. `/resources`
3. `/static`
4. `/public`

Search order: top-down

© luv2code LLC

3rd Party CSS Libraries - Bootstrap

- Local Installation
- Download Bootstrap file(s) and add to **`/static/css`** directory

```
<head>
  … …

  <!-- reference CSS file -->
  <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}" />

</head>
```

© luv2code LLC

3rd Party CSS Libraries - Bootstrap

- Remote Files

```
<head>
  … …

  <!-- reference CSS file -->
  <link rel="stylesheet"
       href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" />

  … …
</head>
```


© luv2code LLC

Spring MVC
Behind the Scenes

© luv2code LLC

Components of a Spring MVC Application

- A set of web pages to layout UI components
- A collection of Spring beans (controllers, services, etc…)
- Spring configuration (XML, Annotations or Java)

© luv2code LLC

How Spring MVC Works Behind the Scenes

© luv2code LLC

Spring MVC Front Controller

- Front controller known as **DispatcherServlet**
  - Part of the Spring Framework
  - Already developed by Spring Dev Team

- You will create
  - Model objects (orange)
  - View templates (dark green)
  - Controller classes (yellow)

© luv2code LLC

Controller

- Code created by developer
- Contains your business logic
  - Handle the request
  - Store/retrieve data (db, web service…)
  - Place data in model
- Send to appropriate view template

© luv2code LLC

Model

- Model: contains your data

- Store/retrieve data via backend systems
  - database, web service, etc…
  - Use a Spring bean if you like

- Place your data in the model
  - Data can be any Java object/collection

© luv2code LLC

View Template

- Spring MVC is flexible
  - Supports many view templates
- Recommended: Thymeleaf
- Developer creates a page
  - Displays data

© luv2code LLC

View Template (more)

- Other view templates supported
  - Groovy, Velocity, Freemarker, etc…
- For details, see:

www.luv2code.com/spring-mvc-views

© luv2code LLC

Reading Form Data with Spring MVC

© luv2code LLC

High Level View

helloworld-form.html

What's your name? Submit Query

helloworld.html

Hello World of Spring!

Student name: John Doe

© luv2code LLC

Application Flow

© luv2code LLC

Application Flow

Request Mapping

/showForm

HelloWorld Controller

helloworld-form.html

What's your name? Submit Query

---

helloworld-form.html

What's your name? Submit Query

Request Mapping

/processForm

HelloWorld Controller

helloworld.html

Hello World of Spring!

Student name: John Doe

© luv2code LLC

Controller Class

```
@Controller
public class HelloWorldController {

   // need a controller method to show the initial HTML form

   @RequestMapping("/showForm")
   public String showForm() {
      return "helloworld-form";
   }

   // need a controller method to process the HTML form

   @RequestMapping("/processForm")
   public String processForm() {
      return "helloworld";
   }
}
```


© luv2code LLC

Development Process

Step-By-Step

1. **Create Controller class**

2. **Show HTML form**

   a. Create controller method to show HTML Form

   b. Create View Page for HTML form

3. **Process HTML Form**

   a. Create controller method to process HTML Form

   b. Develop View Page for Confirmation

© luv2code LLC

Adding Data to Spring Model

© luv2code LLC

Focus on the Model

© luv2code LLC

Spring Model

- The **Model** is a container for your application data

- In your Controller
  - You can put anything in the **model**
  - strings, objects, info from database, etc…

- Your View page can access data from the **model**

© luv2code LLC

Code Example

- We want to create a new method to process form data
- Read the form data: student's name
- Convert the name to upper case
- Add the uppercase version to the model

© luv2code LLC

Passing Model to your Controller

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {

    // read the request parameter from the HTML form
    String theName = request.getParameter("studentName");

    // convert the data to all caps
    theName = theName.toUpperCase();

    // create the message
    String result = "Yo! " + theName;

    // add message to the model
    model.addAttribute("message", result);

    return "helloworld";
}
```


© luv2code LLC

View Template - Thymeleaf

```
<html><body>

Hello World of Spring!
...

The message: <span th:text="${message}" />

</body></html>
```


© luv2code LLC

Adding more data to your Model

```
// get the data
//
String result = ...
List<Student> theStudentList = ...
ShoppingCart theShoppingCart = ...

// add data to the model
//
model.addAttribute("message", result);

model.addAttribute("students", theStudentList);

model.addAttribute("shoppingCart", theShoppingCart);
```


© luv2code LLC

Reading HTML Form Data with @RequestParam Annotation

© luv2code LLC

Code Example

- We want to create a new method to process form data
- Read the form data: student's name
- Convert the name to upper case
- Add the uppercase version to the model

© luv2code LLC

Instead of using HttpServletRequest

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {

   // read the request parameter from the HTML form
   String theName = request.getParameter("studentName");

   ...
}
```


© luv2code LLC

Bind variable using @RequestParam Annotation

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(
        @RequestParam("studentName") String theName,
        Model model) {

   // now we can use the variable: theName

}
```

Behind the scenes:

Spring will read param from request: studentName

Bind it to the variable: theName

© luv2code LLC

@GetMapping and @PostMapping

© luv2code LLC

HTTP Request / Response

© luv2code LLC

Most Commonly Used HTTP Methods

Method	Description
GET	Requests data from given resource
POST	Submits data to given resource
others	…

© luv2code LLC

Sending Data with GET method

```
<form th:action="@{/processForm}" method="GET" ...>
  ...
</form>
```

- Form data is added to end of URL as name/value pairs
  - **theUrl?field1=value1&field2=value2…**

© luv2code LLC

Handling Form Submission

```
@RequestMapping("/processForm")
public String processForm(...) {
  ...
}
```

- This mapping handles **ALL HTTP** methods
  - GET, POST, etc …

© luv2code LLC

Constrain the Request Mapping - GET

```
@RequestMapping(path="/processForm", method=RequestMethod.GET)
public String processForm(...) {
  ...
}
```

- This mapping **ONLY** handles **GET** method
- Any other HTTP REQUEST method will get rejected

© luv2code LLC

Annotation Short-Cut

```
@GetMapping("/processForm")
public String processForm(...) {
  ...
}
```

- @GetMapping: this mapping **ONLY** handles **GET** method
- Any other HTTP REQUEST method will get rejected

© luv2code LLC

Sending Data with POST method

```
<form th:action="@{/processForm}" method="POST" ...>
  ...
</form>
```

- Form data is passed in the body of HTTP request message

© luv2code LLC

Sending Data with POST method

© luv2code LLC

Constrain the Request Mapping - POST

```
@RequestMapping(path="/processForm", method=RequestMethod.POST)
public String processForm(...) {
  ...
}
```

- This mapping **ONLY** handles **POST** method
- Any other HTTP REQUEST method will get rejected

© luv2code LLC

Annotation Short-Cut

```
@PostMapping("/processForm")
public String processForm(...) {
  ...
}
```

- **@PostMapping:** This mapping **ONLY** handles **POST** method
- Any other HTTP REQUEST method will get rejected

© luv2code LLC

Well …. which one???

GET

- Good for debugging
- Bookmark or email URL
- Limitations on data length

POST

- Can't bookmark or email URL
- No limitations on data length
- Can also send binary data

© luv2code LLC

Spring MVC Form Tag - Text Field

© luv2code LLC

Review HTML Forms

- HTML Forms are used to get input from the user

© luv2code LLC

Data Binding

- Spring MVC forms can make use of data binding
- Automatically setting / retrieving data from a Java object / bean

© luv2code LLC

Big Picture

© luv2code LLC

Showing Form

In your Spring Controller

- Before you show the form, you must add a ***model attribute***

- This is a bean that will hold form data for the ***data binding***

© luv2code LLC

Show Form - Add Model Attribute

Code snippet from Controller

```
@GetMapping("/showStudentForm")
public String showForm(Model theModel) {

  theModel.addAttribute("student", new Student());

  return "student-form";
}
```


© luv2code LLC

Setting up HTML Form - Data Binding

Name of model attribute

```
<form th:action="${/processStudentForm}" th:object="${student}" method="POST">

  First name: <input type="text"  th:field="*{firstName}" />

  <br><br>

  Last name: <input type="text"  th:field="*{lastN

  <br><br>

  <input type="submit" value="Submit" />

</form>
```

```
@GetMapping("/showStudentForm")
public String showForm(Model theModel) {

  theModel.addAttribute("student", new Student());

  return "student-form";
}
```


© luv2code LLC

Setting up HTML Form - Data Binding

```
<form th:action="${/processStudentForm}" th:object="${student}" method="POST">

  First name: <input type="text"  th:field="*{firstName}" />

  <br><br>

  Last name: <input type="text"  th:field="*{lastName}" />

  <br><br>

  <input type="submit" value="Submit" />

</form>
```

*{ … } is shortcut syntax for: ${student.firstName}

*{ … } is shortcut syntax for: ${student.lastName}

© luv2code LLC

When Form is Loaded … fields are populated

```
<form th:action="${/processStudentForm}" th:object="${student}" method="POST">

   First name: <input type="text"  th:field="*{firstName}" />

   <br><br>

   Last name: <input type="text"  th:field="*{lastName}" />

   <br><br>

   <input type="submit" value="Submit" />

</form>
```

When form is **loaded**,
Spring MVC will read student
from the model,
then call:

student.getFirstName()
…
student.getLastName()

© luv2code LLC

When Form is submitted … calls setter methods

```
<form th:action="${/processStudentForm}" th:object="${student}" method="POST">

   First name: <input type="text"  th:field="*{firstName}" />

   <br><br>

   Last name: <input type="text"  th:field="*{lastName}" />

   <br><br>

   <input type="submit" value="Submit" />

</form>
```

When form is **submitted**,
Spring MVC will
create a **new** Student instance
and add to the model,
then call:

student.setFirstName(…)
…
student.setLastName(…)

© luv2code LLC

Handling Form Submission in the Controller

Code snippet from Controller

```
@PostMapping("/processStudentForm")
public String processForm(@ModelAttribute("student") Student theStudent) {

   // log the input data
   System.out.println("theStudent: " + theStudent.getFirstName()
                              + " " + theStudent.getLastName());

   return "student-confirmation";
}
```


© luv2code LLC

Confirmation page

```
<html>

<body>

  The student is confirmed: <span th:text="${student.firstName} + ' ' + ${student.lastName}" />

</body>

</html>
```

Calls student.getFirstName()

Calls student.getLastName()

The student is confirmed: John Doe

© luv2code LLC

Pulling It All Together

student-form.html

Student

Student Controller

Student

student-confirmation.html

© luv2code LLC

Development Process

Step-By-Step

1. Create Student class
2. Create Student controller class
3. Create HTML form
4. Create form processing code
5. Create confirmation page

© luv2code LLC

Spring MVC Form - Drop Down List

© luv2code LLC

Review of HTML <select> Tag

```
<select name="country">
    <option value="Brazil">Brazil</option>
    <option value="France">France</option>
    <option value="Germany">Germany</option>
    <option value="India">India</option>
    ...
</select>
```

BR
FR
DE
IN

Value sent during form submission

Displayed to user

© luv2code LLC

Thymeleaf and <select> tag

```
<select th:field="*{country}">

    <option th:value="Brazil">Brazil</option>
    <option th:value="France">France</option>
    <option th:value="Germany">Germany</option>
    <option th:value="India">India</option>

</select>
```

Value sent during form submission

Displayed to user

© luv2code LLC

Pulling It All Together

© luv2code LLC

Development Process

Step-By-Step

1. Update HTML form

2. Update Student class - add getter/setter for new property

3. Update confirmation page

© luv2code LLC

Spring MVC Form - Radio Buttons

© luv2code LLC

Radio Buttons

© luv2code LLC

Code Example

```
Favorite Programming Language:

<input type="radio" th:field="*{favoriteLanguage}" th:value="Go">Go</input>
<input type="radio" th:field="*{favoriteLanguage}" th:value="Java">Java</input>
<input type="radio" th:field="*{favoriteLanguage}" th:value="Python">Python</input>
```

Binding to property on Student object

Value sent during form submission

Displayed to user

© luv2code LLC

Pulling It All Together

© luv2code LLC

Development Process

Step-By-Step

1. Update HTML form

2. Update Student class - add getter/setter for new property

3. Update confirmation page

© luv2code LLC

Spring MVC Forms - Check Box

© luv2code LLC

Check Box - Pick Your Favorite Operating Systems

Favorite Operating Systems: ☐ Linux ☐ macOS ☐ Microsoft Windows

Submit

© luv2code LLC

Code Example

```
<input type="checkbox" th:field="*{favoriteSystems}" th:value="Linux">Linux</input>
<input type="checkbox" th:field="*{favoriteSystems}" th:value="macOS">macOS</input>
<input type="checkbox" th:field="*{favoriteSystems}"
                       th:value="''Microsoft Windows''">Microsoft Windows</input>
```


© luv2code LLC

Pulling It All Together

© luv2code LLC

Development Process

Step-By-Step

1. Update HTML form

2. Update Student class - add getter/setter for new property

3. Update confirmation page

© luv2code LLC

Spring MVC Form Validation

© luv2code LLC

The Need for Validation

Check the user input form for

required fields

valid numbers in a range

valid format (postal code)

custom business rule

© luv2code LLC

Java's Standard Bean Validation API

- Java has a standard Bean Validation API
- Defines a metadata model and API for entity validation
- Spring Boot and Thymeleaf also support the Bean Validation API

http://www.beanvalidation.org

© luv2code LLC

Bean Validation Features

Validation Feature
required
validate length
validate numbers
validate with regular expressions
custom validation

© luv2code LLC

Validation Annotations

Annotation	Description
@NotNull	Checks that the annotated value is not null
@Min	Must be a number >= value
@Max	Must be a number <= value
@Size	Size must match the given size
@Pattern	Must match a regular expression pattern
@Future / @Past	Date must be in future or past of given date
others …	

© luv2code LLC

Our Road Map

1. set up our development environment
2. required field
3. validate number range: min, max
4. validate using regular expression (regexp)
5. custom validation

© luv2code LLC

Spring MVC Form Validation
Required Fields

© luv2code LLC

Required Fields

Fill out the form. Asterisk (*) means required.

First name:

Last name (*):

Submit

Fill out the form. Asterisk (*) means required.

First name:

Last name (*): is required

Submit

© luv2code LLC

Pulling It All Together

customer-form.html

Customer

Customer Controller

Customer

customer-confirmation.html

The customer is confirmed: John Doe

© luv2code LLC

Development Process

Step-By-Step

1. Create Customer class and add validation rules

2. Add Controller code to show HTML form

3. Develop HTML form and add validation support

4. Perform validation in the Controller class

5. Create confirmation page

© luv2code LLC

Step 1: Create Customer class and add validation rules

File: Customer.java

```
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

public class Customer {

    private String firstName;

    @NotNull(message = "is required")
    @Size(min=1, message = "is required")
    private String lastName = “”;

    // getter/setter methods …

}
```

Validation rules

© luv2code LLC

Step 2: Add Controller code to show HTML form

File: CustomerController.java

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.ui.Model;

@Controller
public class CustomerController {

    @GetMapping("/")
    public String showForm(Model theModel) {

        theModel.addAttribute("customer", new Customer());

        return "customer-form";
    }

   …

}
```

Model allows us to share information between Controllers and view pages (Thymeleaf)

name

value

© luv2code LLC

Step 3: Develop HTML form and add validation support

Where to submit form data

Model attribute name

File: customer-form.html

```
<form th:action="@{/processForm}" th:object="${customer}" method="POST">

    First name: <input type="text" th:field="*{firstName}" />

    <br><br>

    Last name (*): <input type="text" th:field="*{lastName}" />

    <!-- Show error message (if present) -->
    <span th:if="${#fields.hasErrors('lastName')}"
          th:errors="*{lastName}"
          class="error"></span>

    <br><br>

    <input type="submit" value="Submit" />

</form>
```

Property name from Customer class

Property name from Customer class

First name:

Last name (*): is required

Submit

© luv2code LLC

Step 4: Perform validation in Controller class

File: CustomerController.java

```
...

@PostMapping("/processForm")
public String processForm(
        @Valid @ModelAttribute("customer") Customer theCustomer,
        BindingResult theBindingResult) {

    if (theBindingResult.hasErrors()) {
        return "customer-form";
    }
    else {
        return "customer-confirmation";
    }
}
```

Model attribute name

Tell Spring MVC to perform validation

The results of validation

© luv2code LLC

Step 5: Create confirmation page

File: customer-confirmation.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<body>

The customer is confirmed: <span th:text="${customer.firstName + ' ' + customer.lastName}" />

</body>

</html>
```


© luv2code LLC

Spring MVC Validation
@InitBinder

© luv2code LLC

White Space

- Our previous example had a problem with white space
  - **Last name** field with all whitespace **passed** … YIKES!
  - Should have **failed**!
- We need to trim whitespace from input fields

© luv2code LLC

@InitBinder

Advanced

- **@InitBinder** annotation works as a pre-processor
- It will pre-process each web request to our controller
- Method annotated with **@InitBinder** is executed

© luv2code LLC

@InitBinder

- We will use this to trim Strings
  - Remove leading and trailing white space
- If String only has white spaces … trim it to **null**
- Will resolve our validation problem … whew :-)

© luv2code LLC

Register Custom Editor in Controller

CustomerController.java

```
...

@InitBinder
public void initBinder(WebDataBinder dataBinder) {

    StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);

    dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);
}

...
```


© luv2code LLC

Spring MVC Validation
Number Range: @Min and @Max

© luv2code LLC

Validate a Number Range

- Add a new input field on our form for: **Free Passes**

- User can only enter a range: 0 to 10

© luv2code LLC

Development Process

Step-By-Step

1. Add validation rule to Customer class

2. Display error messages on HTML form

3. Perform validation in the Controller class

4. Update confirmation page

© luv2code LLC

Step 1: Add validation rule to Customer class

```
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.Max;

public class Customer {

   @Min(value=0, message="must be greater than or equal to zero")
   @Max(value=10, message="must be less than or equal to 10")
   private int freePasses;

   // getter/setter methods

}
```


© luv2code LLC

Development Process

Step-By-Step

1. Add validation rule to Customer class

2. Display error messages on HTML form

3. Perform validation in the Controller class

4. Update confirmation page

© luv2code LLC

Spring MVC Validation
Regular Expressions

© luv2code LLC

Regular Expressions

Advanced

- A sequence of characters that define a search pattern
  - This pattern is used to find or match strings

- Regular Expressions is like its own language (advanced topic)
  - I will assume you already know about regular expressions

- If not, then plenty of free tutorials available
  - https://docs.oracle.com/javase/tutorial/essential/regex/

© luv2code LLC

Validate a Postal Code

- Add a new input field on our form for: **Postal Code**
- User can only enter 5 chars / digits
- Apply ***Regular Expression***

© luv2code LLC

Development Process

Step-By-Step

1. Add validation rule to Customer class

2. Display error messages on HTML form

3. Update confirmation page

© luv2code LLC

Step 1: Add validation rule to Customer class

Advanced

```
import jakarta.validation.constraints.Pattern;

public class Customer {

   @Pattern(regexp="^[a-zA-Z0-9]{5}", message="only 5 chars/digits")
   private String postalCode;

   // getter/setter methods

}
```


© luv2code LLC

Spring MVC Validation
Make an Integer Field Required

© luv2code LLC

Spring MVC Validation
Handle String Input for Integer Fields

© luv2code LLC

Development Process

Step-By-Step

1. Create custom error message

- src/resources/messages.properties

© luv2code LLC

Spring MVC Validation
Custom Validation

© luv2code LLC

Custom Validation Demo

First name:

Last name:

Course Code:

Submit

© luv2code LLC

Custom Validation

- Perform custom validation based on your business rules
  - Our example: Course Code must start with “LUV”
- Spring MVC calls our custom validation
- Custom validation returns boolean value for pass/fail (true/false)

© luv2code LLC

Create a custom Java Annotation … from scratch

Advanced

- So far, we've used predefined validation rules: @Min, @Max, …
- For custom validation … we will create a **Custom Java Annotation**
  - **@CourseCode**

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```


© luv2code LLC

Development Process

Step-By-Step

1. Create custom validation rule

2. Add validation rule to Customer class

3. Display error messages on HTML form

4. Update confirmation page

© luv2code LLC

Development Process - Drill Down

Step-By-Step

1. Create custom validation rule
    a. Create **@CourseCode** annotation
    b. Create **CourseCodeConstraintValidator**

© luv2code LLC

Step 1a: Create @CourseCode annotation

Advanced

Usage Example

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```


© luv2code LLC

Step 1a: Create @CourseCode annotation

Advanced

```
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD } )
@Retention(RetentionPolicy.RUNTIME)
public @interface CourseCode {



  ...
}
```

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```


© luv2code LLC

Step 1a: Create @CourseCode annotation

Advanced

```
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD } )
@Retention(RetentionPolicy.RUNTIME)
public @interface CourseCode {

   // define default course code
   public String value() default "LUV";



   ...
}
```

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```


© luv2code LLC

Step 1a: Create @CourseCode annotation

Advanced

```
@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";

   ...
}
```

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```


© luv2code LLC

Step 1b: Create CourseCodeConstraintValidator

Advanced

```
import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class CourseCodeConstraintValidator
    implements ConstraintValidator<CourseCode, String> {

    private String coursePrefix;

    @Override
    public void initialize(CourseCode theCourseCode) {
        coursePrefix = theCourseCode.value();
    }

    @Override
    public boolean isValid(String theCode,
                           ConstraintValidatorContext theConstraintValidatorContext) {

        boolean result;

        if (theCode != null) {
            result = theCode.startsWith(coursePrefix);
        }
        else {
            result = true;
        }

        return result;
    }
}
```


© luv2code LLC