

# Lab – Spring MVC App using Spring Boot

- You should be able to see the product list like below :



## Product List

[Create New Product](#)

Product ID	Name	Brand	Made In	Price	Actions
1	laptop	Mac	USA	400.0	<a href="#"><u>Edit</u></a> <a href="#"><u>Delete</u></a>
2	Laptop	Dell	China	200.0	<a href="#"><u>Edit</u></a> <a href="#"><u>Delete</u></a>

# Lab – Spring MVC App using Spring Boot

- Let's continue and implement 'Edit' feature.
- In products.html we have already defined that on click of 'Edit' button, request should go to '/edit/\${id}' in our controller.

```
<td>  
    <a th:href="@{'/edit/' + ${product.id}}">Edit</a>  
        &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
    <a th:href="@{'/delete/' + ${product.id}}">Delete</a>  
</td>
```

# Lab – Spring MVC App using Spring Boot

- Create a mapping in the controller for edit like below.

```
@GetMapping("/edit/{id}")
public ModelAndView showEditProductPage(@PathVariable(name = "id") long id) {
    ModelAndView modelAndView = new ModelAndView("edit-product");
    Product product = productRepository.findById(id).get();
    modelAndView.addObject("product", product);
    return modelAndView;
}
```

# Lab – Spring MVC App using Spring Boot

- Couple of things to note in the code we just added.
- We can create parameterized urls like this.  
`@GetMapping("/edit/{id}")`
- We can read the value of parameterized variable using *@PathVariable*.
- *productRepository.findById* is a standard spring-data method to fetch record by ID field.

# Lab – Spring MVC App using Spring Boot

- Now we have controller ready to accept 'edit' request.
- Let's create a view *edit-product.html* inside 'templates' folder for our edit screen.

# edit-product.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8" />
    <title>Edit Product</title>
</head>
<body>

</body>

</html>
```

# edit-product.html (inside <body> tag)

```
<div align="center">
  <h1>Edit Product</h1>
  <br />
  <form action="#" th:action="@{/save}" th:object="${product}" method="post">
    <table border="0" cellpadding="10">
      <tr>
        <td>Product ID:</td>
        <td>
          <input type="text" th:field="*{id}" readonly="readonly" />
        </td>
      </tr>
    </table>
  </form>
</div>
```

# edit-product.html (inside <body> tag)

```
<tr>
  <td>Product Name:</td>
  <td>
    <input type="text" th:field="*{name}" />
  </td>
</tr>
<tr>
  <td>Brand:</td>
  <td><input type="text" th:field="*{brand}" /></td>
</tr>
<tr>
  <td>Made In:</td>
  <td><input type="text" th:field="*{madein}" /></td>
</tr>
```



# edit-product.html (inside <body> tag)

```
<tr>
    <td>Price:</td>
    <td><input type="text" th:field="*{price}" /></td>
</tr>
<tr>
    <td colspan="2"><button type="submit">Save</button> </td>
</tr>
</table>
</form>
</div>
```

# Lab – Spring MVC App using Spring Boot

Hit *Restart* button.

Head over to  
browser and *refresh*.  
Hit *Edit* button for  
any one product.

# Lab – Spring MVC App using Spring Boot

- You should be able to see edit page for the product you selected.
- All the saved details of the product should be filled in.

## Edit Product

Product ID:	<input type="text" value="1"/>
Product Name:	<input type="text" value="laptop"/>
Brand:	<input type="text" value="Mac"/>
Made In:	<input type="text" value="USA"/>
Price:	<input type="text" value="400.0"/>

Save

# Lab – Spring MVC App using Spring Boot



Edit some of the fields and Hit 'Save'.



You should be redirected to product-list page. Edited details should be reflected.



See the changes in the database to confirm that the edited details really got saved in Db and not just UI.

# Lab – Spring MVC App using Spring Boot

Did you observe that we used same '/save' controller mapping which we used to create new product?

But this time instead of creating new product our code updated the same product.

Why So??

# Lab – Spring MVC App using Spring Boot



**Magic of 'save' method from spring repository.**



**When we call 'save' method on an Entity, it checks for 'id' field in entity.**

If 'id' is *null* it will create new record.

If 'id' is not null then it checks for the record with same 'id' in table.

- If there is record with same 'id' it will get updated.
- If not then new record will be created with 'id' value passed.

# Lab – Spring MVC App using Spring Boot

- Let's Implement 'Delete' feature.
- Update controller with code for “/delete” mapping.

```
@RequestMapping("/delete/{id}")  
public String deleteProduct(@PathVariable(name = "id") long id) {  
    productRepository.deleteById(id);  
    return "redirect:/";  
}
```

# Lab – Spring MVC App using Spring Boot

- In products.html we already have code to send request to our “/delete” mapping.

  |

```
<a th:href="@{'/edit/' + ${product.id}}">Edit</a>
```

```
<a th:href="@{'/delete/' + ${product.id}}">Delete</a>
```

&lt;/td&gt;



# Lab – Spring MVC App using Spring Boot



Restart the app.




Hit *Delete* button for the product you want to remove.



Product should get removed from list.

Bootstrap

The image features a dark purple rectangular background. In the center, there is a white square containing a large, bold, serif letter 'B'. To the right of this square, the words 'Bootstrap Framework' are written in a white, sans-serif font. The background is decorated with faint, light-colored outlines of various electronic devices, including a desktop monitor, a laptop, and a smartphone, arranged in a way that suggests a modern, tech-oriented environment.

**B** Bootstrap Framework

# Bootstrap Framework

- Bootstrap is a powerful front-end framework for faster and easier web development.
- It includes HTML and CSS based design templates for creating common user interface components like forms, buttons, navigations, dropdowns, alerts, modals, tabs, accordions, carousels, tooltips, and so on.

# Making this HTML File a Bootstrap Template

```
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <title>Basic Bootstrap Template</title>
  <!-- Bootstrap CSS file -->
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css">
</head>
<body>
  <h1>Hello, world!</h1>
  <!-- JS files: jQuery first, then Popper.js, then Bootstrap JS -->
  <script src="https://code.jquery.com/jquery-3.5.1.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js"></script>
</body>
```

# Making this HTML File a Bootstrap Template

**And we're all set!** After adding the Bootstrap's CSS and JS files and the required jQuery and Popper.js library, we can begin to develop any site or application with the Bootstrap framework.

The attributes integrity and crossorigin have been added to CDN links to implement Subresource Integrity (SRI). It is a security feature that enables you to mitigate the risk of attacks originating from compromised CDNs, by ensuring that the files your website fetches (from a CDN or anywhere) have been delivered without unexpected or malicious modifications.

# Lab : Add bootstrap to products.html

- Open products.html from our ProductInventory App.
- Add bootstrap to it.
- Restart the app and open in browser.
- You should observe some changes in styling of page.
- Bootstrap started showing it's magic.

# Lab : Add bootstrap to products.html

<div><div>← → ↻ 🏠 ⓘ localhost:8080</div></div>					
<h2>Product List</h2> <div>Create New Product</div>					
Product ID	Name	Brand	Made In	Price	Actions
1	laptop	Mac	USA	450.0	<a href="#">Edit</a> <a href="#">Delete</a>
3	One Plus 9	OnePlus	China	500.0	<a href="#">Edit</a> <a href="#">Delete</a>

# Creating Containers with Bootstrap

- Containers are the most basic layout element in Bootstrap and are required when using the grid system.
- Containers are basically used to wrap content with some padding.
- They are also used to align the content horizontally center on the page in case of fixed width layout.



# Creating Containers with Bootstrap

- Bootstrap provides three different types containers:
  - `.container`, which has a max-width at each responsive breakpoint.
  - `.container-fluid`, which has 100% width at all breakpoints.
  - `.container-{breakpoint}`, which has 100% width until the specified breakpoint.
- The following table illustrates how each container's max-width changes across each breakpoint.

# Creating Containers with Bootstrap

Class	Extra small <576px	Small ≥576px	Medium ≥768px	Large ≥992px	Extra large ≥1200px
.container	100%	540px	720px	960px	1140px
.container-sm	100%	540px	720px	960px	1140px
.container-md	100%	100%	720px	960px	1140px
.container-lg	100%	100%	100%	960px	1140px
.container-xl	100%	100%	100%	100%	1140px
.container-fluid	100%	100%	100%	100%	100%

# Code along : Let's add *container-fluid* to products.html

- Wrap the content of <body> tag in a <div>.
- Provide .container-fluid class to the div.
- Restart the app and observe changes.

```
<body>
  <div class="container-fluid">
    <h1>Product List</h1>
    <a href="new">Create New Product</a>
    <br/><br/>
    <table border="1" cellpadding="10">
      // removed for brevity
    </table>
  </div>
  <!-- JS files: jQuery first, then Popper.js, then Bootstrap JS -->
  <script src="https://code.jquery.com/jquery-3.5.1.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js"></script>
</body>
```

# Bootstrap Buttons

- Bootstrap includes several predefined button styles, each serving its own semantic purpose, with a few extras thrown in for more control.



```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>

<button type="button" class="btn btn-link">Link</button>
```

Copy

# Bootstrap Buttons

- Let's style our 'Create New Product' link on products.html.

- Here is our link:

```
<a href="new">Create New Product</a>
```

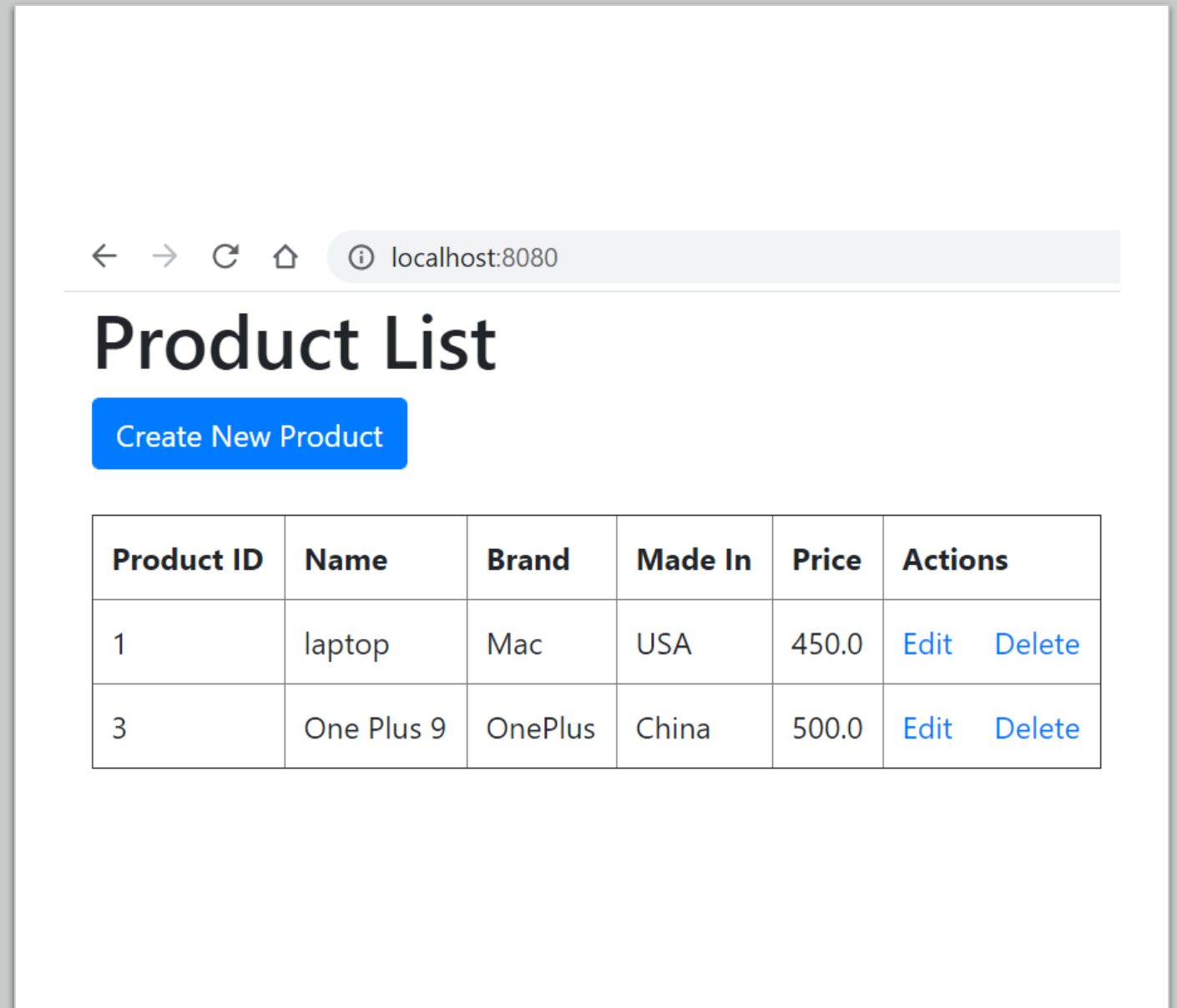
- Let's add bootstrap classes to it.

```
<a class="btn btn-primary" href="new">Create New Product</a>
```

- Restart the app and open in browser.

# Bootstrap Buttons

- You should be able to see our link changed to nice looking button.



# Lab : Style 'Edit' and 'Delete' links

- In products.html, style 'Edit' link to look like blue button.
- And style 'Delete' link to look like Red button.
- You can refer bootstrap official website.
- Here is the link for bootstrap components :
  - <https://getbootstrap.com/docs/4.6/components/alerts/>

# Lab : Style 'Edit' and 'Delete' links

Product ID	Name	Brand	Made In	Price	Actions	
1	laptop	Mac	USA	450.0	Edit	Delete
3	One Plus 9	OnePlus	China	500.0	Edit	Delete



# Creating a Simple Navbar with Bootstrap

- You can use the Bootstrap navbar component to create responsive navigation header for your website or application.
- These responsive navbar initially collapsed on devices having small viewports like cell-phones but expand when user click the toggle button.
- However, it will be horizontal as normal on the medium and large devices like laptop or desktop.
- You can also create different variations of the navbar such as navbars with dropdown menus and search boxes as well as fixed positioned navbar with much less effort.

# basic.html with navbar

```
<nav class="navbar navbar-expand-md navbar-dark bg-dark">
  <a href="#" class="navbar-brand">Brand</a>
  <button type="button" class="navbar-toggler" data-toggle="collapse" data-target="#navbarCollapse">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarCollapse">
    <div class="navbar-nav">
      <a href="#" class="nav-item nav-link active">Home</a>
      <a href="#" class="nav-item nav-link">Profile</a>
      <a href="#" class="nav-item nav-link">Messages</a>
      <a href="#" class="nav-item nav-link disabled" tabindex="-1">Reports</a>
    </div>
    <div class="navbar-nav ml-auto">
      <a href="#" class="nav-item nav-link">Login</a>
    </div>
  </div>
</nav>
```

## Lab : Add Navbar to our products.html

<div>← → ↻ 🏠 ⓘ localhost:8080</div> <div>ProductInventory Home About Us</div>					
<div>Create New Product</div>					
Product ID	Name	Brand	Made In	Price	Actions
1	laptop	Mac	USA	450.0	<div>EditDelete</div>
3	One Plus 9	OnePlus	China	500.0	<div>EditDelete</div>



## Layout in Thymeleaf

---

- Thymeleaf Standard Layout System offers page fragment inclusion that is similar to *JSP includes*, with some important improvements over them.

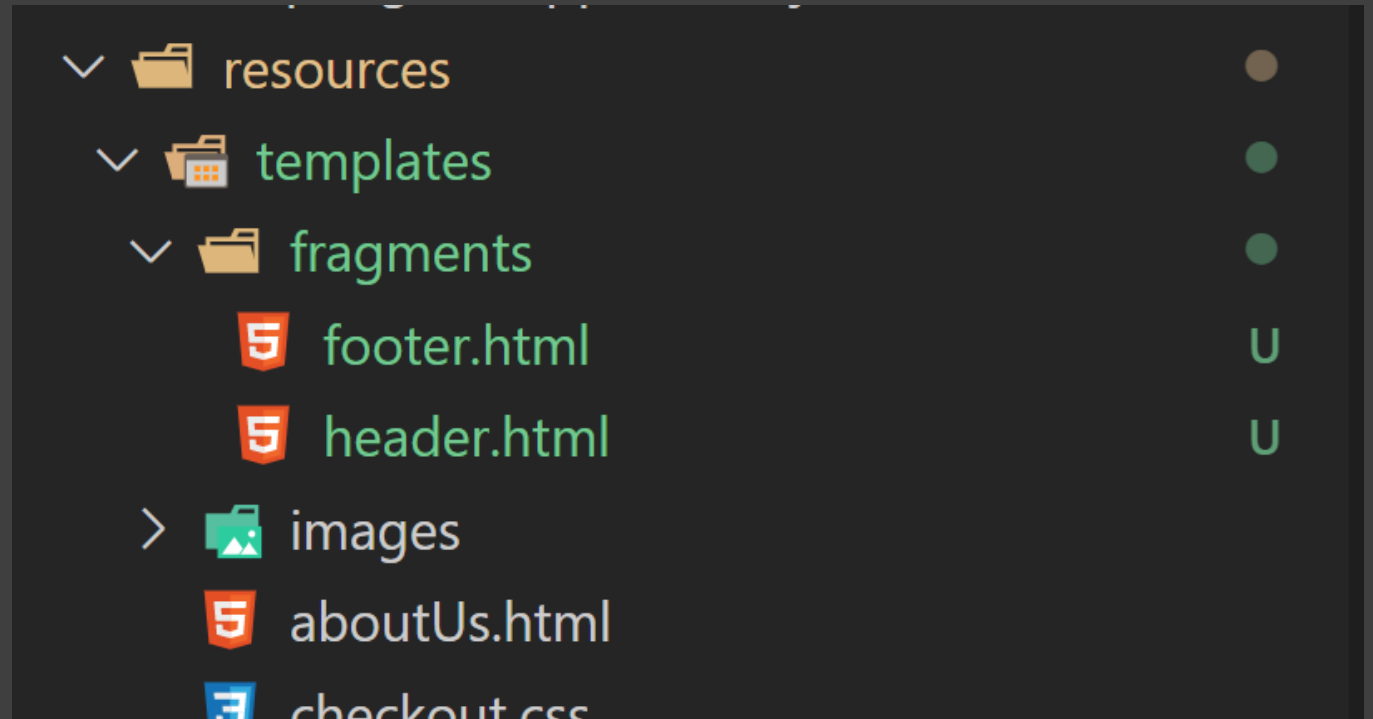
```
@GetMapping("/")
public String viewHomePage(Model model) {
    List<Product> products = productRepository.findAll();
    model.addAttribute("products", products);
    return "index";
}
```

## Controller

First, we'll tell the controller what view we want to use, as usual.

# Fragments

I have added my fragments to a special folder to have some structure.



# Content

Let's add content to header.html and footer.html and mark the fragments we'd like to use.

```
rties M    5 header.html U X    5 index.html M    ☕ ProductController.java  
in > resources > templates > fragments > 5 header.html  
<header th:fragment="header">This is a header</header>  
|
```

```
.properties M X    5 header.html U    5 index.html M    ☕ ProductController.java  
c > main > resources > templates > fragments > 5 footer.html  
1 <footer th:fragment="footer">This is a footer.</footer>  
2 |
```

```
11 </head>
12
13 <div th:replace="fragments/header :: header"></div>
14
15 <h1>Content</h1>
16
17 <div th:replace="fragments/footer :: footer"></div>
18
19 </body>
```

## Add fragments

- Finally, lets add the fragments to our content.





localhost:8080



Appar



Kolla



Verktug



Övr

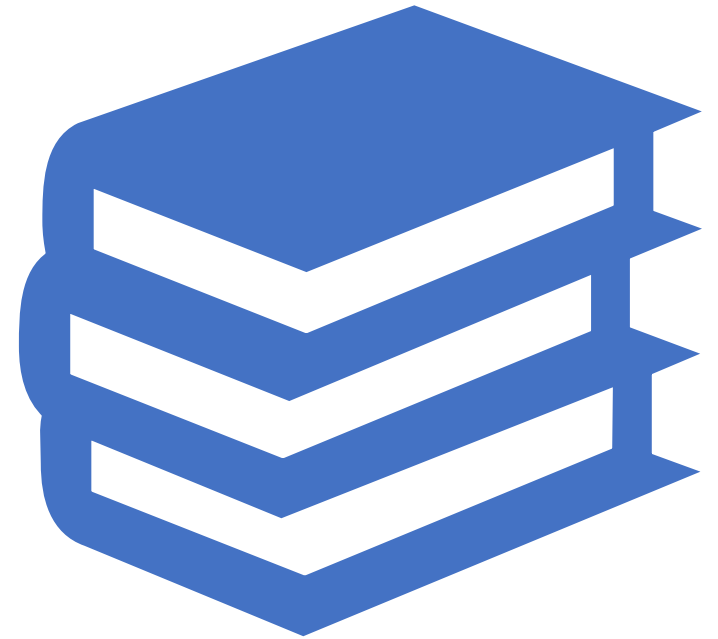
This is a header

# Content

This is a footer.

Read more

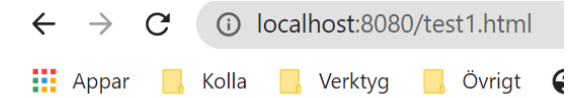
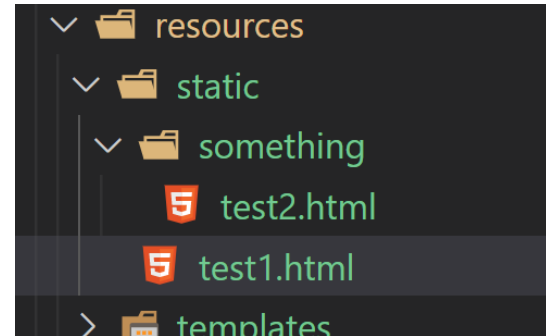
- <https://www.thymeleaf.org/doc/articles/layouts.html>



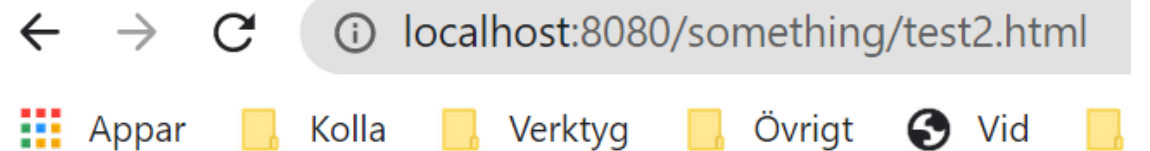
# Images and parameters

- So, in your html file, you have your productCode.
- Can we use it for our product images?
- First of all, images, css files, js files etc are static and don't need to be parse/compiled by Java.
- Static files can be stored on a number of locations.
- Spring Boot comes with a pre-configured implementation of *ResourceHttpRequestHandler* to facilitate serving static resources.
- By default, this handler serves static content from any of /static, /public, /resources, and /META-INF/resources directories that are on the classpath. Since src/main/resources is typically on the classpath by default, we can place any of these directories there.
- For example, if we put an *about.html* file inside the /static directory in our classpath, then we can access that file via <http://localhost:8080/about.html>.
- Similarly, we can achieve the same result by adding that file in other mentioned directories.

# Images and parameters



## Test 1



## Test 2

# Images and parameters

- Second of all, when we loop through our objects, we have access to each property of that object, like the productCode.
- Do you remember that I suggested that you'd name your images <productCode>.png?

```
<tr th:each="product : ${products}">
  <td></td>
  <td th:text="${product.productname}">Name</td>
</tr>
```

# Query Limits

If we want to get just a few results rather than the full result set in MSSQL server, we use TOP.

```
SELECT TOP 10 * FROM products;
```

- In MySQL we use LIMIT.

```
SELECT * FROM products LIMIT 10;
```

- In JPA, we have some magical methods that, among other things, let us limit the number of results. We just need to put them in our repository interface.

# Magical methods

- These methods are available in all our repositories.
- You can read more about them here:
- <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.details>

```
@GetMapping("/")
public String viewHomePage(Model model) {
    List<Products> products = productsRepository.findTop10ByProductline("Classic cars");
    List<Products> products2 = productsRepository.findTop10ByOrderByProductcodeAsc();
    model.addAttribute("products", products);
    return "products";
}
```

```
public interface ProductsRepository extends JpaRepository<Products, String> {
    List<Products> findTop10ByProductline(String productline);

    List<Products> findTop10ByOrderByProductcodeAsc();
}
```

# Form Validation

- Form Data Validation is a very common, and rudimentary step in building any web application with user input.
- We want to make sure that certain ranges are respected, and that certain formats are followed.
- For example, we'll want to make sure that the user isn't -345 years old, or that their email address is valid.



# Form Validation

- There are many ways to validate form data - and the method you use depends on your application.
- In general, you'll want to perform client-side validation, as well as server-side validation.
- Client-side validation is faster because you don't have to make a request or reload the page, while server-side validation is safer because it can't be manipulated the same way.

# Form Validation

- We'll cover how to perform form data validation in Spring Boot with Thymeleaf, as the template engine.
- We'll leverage Spring Boot's built-in *Bean Validation API*, which makes this process simple and straightforward.

# Form Validation

- Spring Boot allows us to define validation criteria using annotations.
- In your Domain Model, you can simply annotate fields with the constraints, and it'll enforce them.



# Form Validation

For the validation annotations to work we need to add the following dependency:

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

# Form Validation

- Consider a 'Person' Domain Model that we want to validate.
- Let's say we want following validations :
  - *fullname* should not be empty.
  - *fullname* should atleast be 5 character long.
  - *email* should not be empty.
  - *email* should be in proper format.
  - *age* should not be null.
  - *age* should be minimum 18.

```
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private Long id;

    private String fullName;

    private String email;

    private Integer age;

    // Getters and Setters

}
```

# Form Validation

- Let's annotate our fields with spring annotations to apply validations.

```
@NotEmpty  
@Size(min = 5)  
private String fullName;
```

```
@NotEmpty  
@Email  
private String email;
```

```
@NotNull  
@Min(value = 18)  
private Integer age;
```

# Form Validation

- Now, let's break the annotations we used:
- **@NotEmpty** - is used to constrain a field of type String, Collection, Map, or Array to not be null or empty.
- **@Size([min = x, max = y])** - is used to define the rules for the size of a String, Collection, Map, or Array.
- **@Email** - helps us to validate the string against a regex which defines the structure of a valid email.

# Form Validation

- **@NotNull** - tells Spring the field should not be null, but it can be empty.
- **@Min** and **@Max** are used to specify the limits of a variable. For example, the @Min age could be set to, say, 18.



# Form Validation

- This is Thymeleaf form for our Domain Model.

```
<form th:action="@{/add}" th:object="${person}" method="post" class="form">
  <div class="form-group">
    <label for="fullName">Name</label>
    <input class="form-control" type="text" th:field="*{fullName}" id="fullName" placeholder="Full Name">
    <div class="alert alert-warning" th:if="${#fields.hasErrors('fullName')}" th:errors="*{fullName}"></div>
  </div>
  <div class="form-group">
    <label for="email">Email</label>
    <input class="form-control" type="text" th:field="*{email}" id="email" placeholder="Email">
    <div class="alert alert-warning" th:if="${#fields.hasErrors('email')}" th:errors="*{email}"></div>
  </div>
  <div class="form-group">
    <label for="age">Age</label>
    <input class="form-control" type="text" th:field="*{age}" id="age" placeholder="Age">
    <div class="alert alert-warning" th:if="${#fields.hasErrors('age')}" th:errors="*{age}"></div>
  </div>
  <input type="submit" class="btn btn-success" value="Add User">
</form>
```

# Form Validation

- The fields in our object, *fullName*, *age* and *email* are in the form, signified by *th:field*.
- Since we've got our *th:object=\${person}*, we can refer to this person object by substituting it with a *\** before the fields.
- *\*{fullName}* is the same as *\${person.fullName}*

# Form Validation

- Each input also has a hidden `<div>` which is shown only if the `${#fields.hasErrors()}` calls evaluate to *true*.
- If there aren't errors, this *div* doesn't exist.



# Form Validation

- If there are, the *th:errors* tag lets us specify a message.
- If we simply pass in the field that causes the error, such as *th:errors="\*{age}"*, the default message from the Bean Validator API are used, for that field.

# Form Validation - Controller

- Now, let's make a controller that'll handle a request to save a Person into the database.
- As usual, we'll have a *@GetMapping()* to show the form, and a *@PostMapping* to handle the request.
- In the method signature of the *@PostMapping*, we'll annotate the POJO with *@Valid*.

# Form Validation - Controller

- The *@Valid* annotation triggers the Bean Validator to check if the fields populated in the object conform to the annotations we've used in the class definition.
- If you don't use the *@Valid* annotation, it won't check anything, and even values you might not expect can be populated in the object.

# Form Validation - Controller

- We also pass in a *BindingResult* object. That's the object that will inform us if there are errors in the validation. It's important to place it right after the object that we're validating in the method signature.
- In the code itself, all we're doing is grabbing the boolean returned by *hasErrors()* on the *BindingResult* object. If that returns true, then there are validation errors, and we send the user back to the form.

# Form Validation - Controller

```
@GetMapping("/add")  
public String showAddPersonForm(Person person)  
{  
    return "add-person";  
}
```

```
@PostMapping("/add")  
public String addPerson(@Valid Person person,  
    BindingResult result, Model model) {  
    if (result.hasErrors()) {  
        return "add-person";  
    }  
    repository.save(person);  
    return "redirect:/index";  
}
```



# Form Validation - Controller

- In our case, the *Person person* object is the object that's populated with the inputs of a form.
- If there are any issues with this *Person*, the *fields* attribute will have errors within it, tied to the field that caused the error.



# Lab : Add Form Validation in ProductInventory App

- Add following validation to *Product* domain model in our *ProductInventory* App.
  - *name* field should not be empty and it should be at least 5 characters long.
  - *brand* field should not be empty.
  - *madein* field should not be empty and it should be at least 3 characters long.
  - *price* field should not be null and it should not be negative OR Zero.
- Modify the 'create new product' form to show error messages.
- Refer here for help :
  - <https://www.baeldung.com/javax-validation>

# Lab : Add Form Validation in ProductInventory App

- When user tries to submit form with invalid values :
  - Data should not get saved in the database.
  - User should remain on the same page.
  - Proper Error message should be shown to user at right side of the invalid field.

# Lab : Add Form Validation in ProductInventory App

## Create New Product

Product Name:

size must be between 5 and 2147483647

Brand:

Made In:

size must be between 3 and 2147483647  
must not be empty

Price:

must be greater than 0

# Form Validation - Customizing Error Messages

- To set a custom message for any validation constraint, you can use the *message* option:
- `@NotEmpty(message = "Field can't be empty!")`
- `private String field;`



# Form Validation - Customizing Error Messages

- You can simply write these messages out in your model, like this.
- Though, it's considered good practice to put the validation messages in a properties file, that you can reference.
- This is done since you can bundle all validation messages together and can update them if the models get changed later.





# Form Validation - Customizing Error Messages

- Let's make our *ValidationMessages.properties* file under *src/main/resources*.
- By default, Spring will resolve messages from this file.
- Let's add a property.  
`Size.Person.FullName=The Full Name should have at least 5 characters`



## Form Validation - Customizing Error Messages

- We will then modify the Person model, and supply this property as the message of the @Size annotation.

```
@NotEmpty(message = "The Full Name can't be null")  
@Size(min = 5, message = "{Size.Person.FullName}")  
private String fullName;
```



# Lab : Customize messages in our App

- Create ValidationMessages.properties inside 'src/main/resources'.
- Modify Product.java to read the messages from properties file.