Distributed Applications WS2025/26 (Prof. Dr. Cathrin Möller)

Exercise 3 - 6./10.11.2025

Practical Exercises

Do not hesitate to ask questions or if you want feedback for your coding style.

Please commit and push to GitLab regularly and ensure @*cathrin.moeller* and my two tutors @pinky-jitendra.tinani and @rahul.patil are invited as a reporter.

Learning goals for today

Today we want to learn using the **Thymeleaf templating** engine to render dynamic model data as HTML. Our Controllers will have to be adjusted to return templates rather than JSON objects. Model values will have to be populated.

Our webshop will get multiple views of the same resource (product entity) like a product detail, a catalog page and a shopping cart. This is highlighting that resource data can be re-used for different views. The ***id*** attribute of a model object is often used internally in the template - to be added for requests - while not always visible to the end user. We can use the templating to generate links including the id parameter.

Practical Exercises

1) Ensure you did the exercises from the three previous weeks to have the starter code for this exercise. If not, please revisit those to get a basic SpringBoot / SpringWeb application.

2) The **Model - View - Controller** pattern is a clean separation method that allows to create a dynamic view which holds placeholder for model objects and properties (instead of delivering fully evaluated static HTML). We can use the **Thymeleaf** templating engine (dependency added to our Spring Application) to render HTML templates rather than plain JSON responses for our Controller endpoints.

Our goal is to use the Thymeleaf templating to have a HTML **View** displaying the product properties for the single product in a „**Product**

Detail Page“ and another **View** for the list of products as a „**Catalog Page**“ linking to the product detail pages.

Create one or two new Controllers. We cannot extend our existing ProductController because it returns JSON using the @RestController annotation. Our new controller(s) shall return a Model and a View and will get the annotation **@Controller.**

Names could e.g. be ProductDetailController and ProductCatalogController if you separate those.

One way to get a clean architecture with lots of controllers is adding them to a package /controller within our domain folder /product.

As our application is of demo purpose, even though we want to stay as „RESTful“ as possible, we need to create new URL mappings for retrieving the ProductDetailPage and the ProductCatalogPage. If you used /products earlier, you could use /product now or the other way around. Or you define a prefix in the URLs to distinguish the APIs like /mvc-api/product and /rest-api/product.

Check the lecture slides how to change our Controller endpoints returning a template and populating the Model for the View.

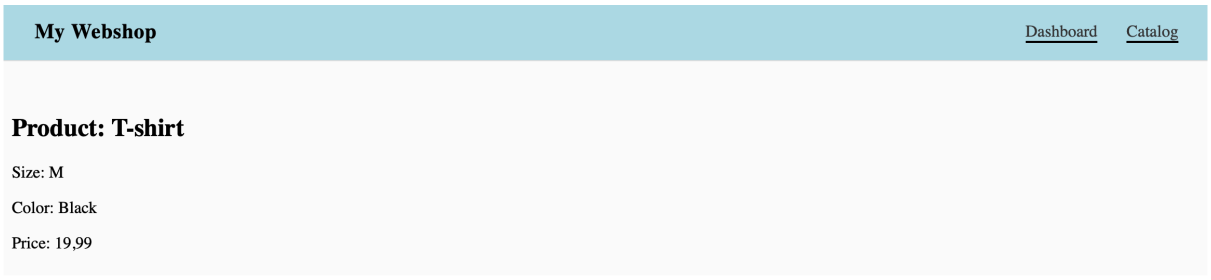
Hint: Template (HTML) files need to be placed under /resources/templates and controllers need to populate a model plus return a view (name of template file without the .html).

Create a header within your HTML with a link or button to navigate back from „Product Detail Page“ to the „Catalog Page“ and a link back to your landing page (dashboard) to not have to rely on browser back.

You could link the catalog page also on your dashboard in order to navigate without typing in the URL.

Feel free to add more hardcoded products to have a bigger catalog.

Results should look similar (could be prettier, feel free to be creative) like this:


My Webshop Dashboard Catalog

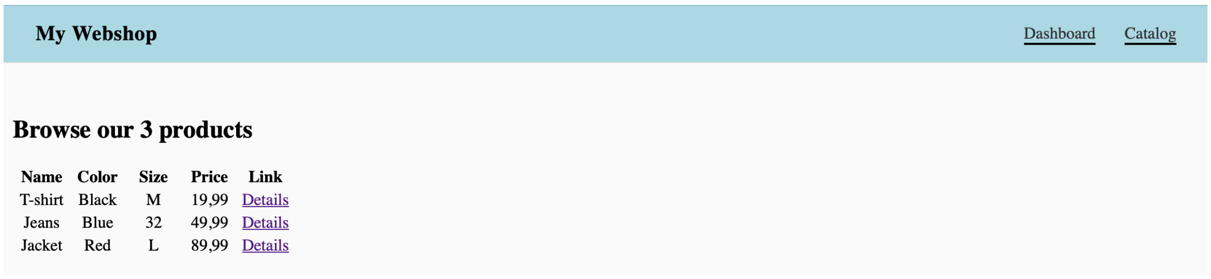
Product: T-shirt

Size: M

Color: Black

Price: 19,99

// Product Detail Page


My Webshop Dashboard Catalog

Browse our 3 products

Name	Color	Size	Price	Link
T-shirt	Black	M	19,99	Details
Jeans	Blue	32	49,99	Details
Jacket	Red	L	89,99	Details

// Catalog Page

3) Add a request parameter based „mode“ *?edit=true/false* that renders the Catalog Page with (or without) delete buttons next to the products. Clicking on any of these delete buttons should trigger a deletion for the selected id and return the remaining catalog (as a model and view reusing the template for the catalog). If you deleted all products, restarting your application can recreate the hardcoded products.

Hints: You can use additional info to the Request Param mapping:

```
@RequestParam(required=false, defaultValue = "Hello")
```

And you can use ***th:if=„${}“*** for conditional rendering in the template.

You will not be able to easily call a DELETE request with the id parameter (would need javascript). So it is ok if you add an „non RESTful“ Endpoint like @GetMapping(„/product-delete/{id}“) for this exercise. Call your ProductService from the Controller which contains logic to remove a product by id from your list of products.

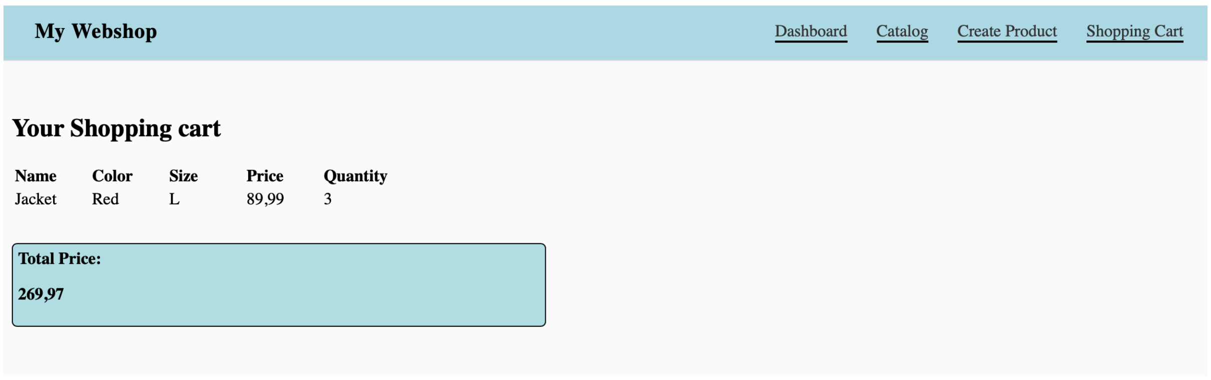
Info: You can also make use of a redirect response in your controller even to views used in other controllers like this:

```
return "redirect:/catalog?edit=true";
```

4) Using MVC and CRUD, we can reuse the same model (entity) and present it in multiple different views. Create a new **ShoppingCartController** and a **ShoppingCartService** as well as a new object class for the model **ShoppingCart.** Create a HTML template for a shopping cart page (View). This page will display (partial) product data combined with further information.

The shopping cart page template view should list all included products with their quantity and show a calculated total price. It also has a header linking to other pages like the catalog page so we can browse further products and add them to the cart.

The result might look similar to this:



Your Shopping cart

Name	Color	Size	Price	Quantity
Jacket	Red	L	89,99	3

Total Price:

269,97

In order to reuse the ShoppingCart across your application you could instantiate a local state in the ShoppingCartService by e.g. creating a new instance of the ShoppingCart in the service constructor. This way you can add and remove items to a singleton ShoppingCart instance.

Hint: Instantiating a shopping cart object you might use a **HashMap** and make all attributes public to ensure the model can be read from the template.

```
public Map<Product, Integer> products = new HashMap<>();
```

Info: We do so for demo purposes, in order to be RESTful, such state would either have to be persisted in the frontend or in a database table for a logged in user account. It works in our case as long as our JavaSpringBoot backend is used by a single user locally. If we would deploy the application to the cloud and the server would be retrieved by more than one client, a global singleton state across clients would not suffice.

In the **ShoppingCartController**, create **endpoints** for

- Viewing the complete shopping cart content on a Thymeleaf template page. Response is the cart View. (@GetMapping(„/cart“))

- Adding products to the cart (by id) (GET and un-RESTful URL naming allowed for simplification - response is a redirect to cart view) (@GetMapping(„/cart-add/{id}“))

We need an „add to cart“ button on the ProductDetail page to call the endpoint that adds items to the cart (including the id of the product selected). The ShoppingCartController then needs to call the ShoppingCartService which might need to call the ProductService to lookup the Product by the **@PathVariable** id.

Hint: For the add to cart button consider using the Thymeleaf feature th:href which allows creating a dynamic link (e.g. with an id from the product model).

Hint: You need this snippet for the Thymeleaf template to parse a HashMap for its entries (the entry.key is the product and the entry.value is the quantity):

```
<tr th:each="entry: ${cart.products.entrySet()}">
    <td th:text="${entry.key.name}">ProductName</td>
    <td th:text="${entry.value}">Quantity</td>
```

Info: If you want to avoid copy pasting the same <header> across multiple Thymeleaf templates, there is a mechanism to include „fragments“:

```
<header th:replace="fragments/header :: header"></header>
```

(assuming you have a template file /templates/fragments.header.html with a header node)