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

Exercise 4 - 13/17.11.2025

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 continue coding on our webshop application with Spring Boot. If you missed previous weeks, please do all the missing exercises that belong to „Products“ and the Webshop Part of a JavaSpring Boot application.

We want to practice using the **Layered** Architecture Pattern and discover the pattern of using **Data Transfer Objects** (DTOs) for the Model.

We can orchestrate the code of a bigger SpringBoot application defining an arbitrary number of layers.

A common structure could be Controller->Facades->Services.

The **Controller Layer** would be handling requests and asking the next layer for actions and data. The **Facade Layer** would encapsulate the logic which sub services are required to combine data from different services or trigger certain actions. A **Service Layer** would be interacting with Repositories persisting data or (simplified if we have no Repository yet) hold our hardcoded data and maintain the state.

Usually the model in a template is also a combination of more than one Model class. We use **DTOs** for that. A ProductDetailDTO might combine the Product.java information with utilities like boolean isSoldOut and Integer stock.

Practical Exercises

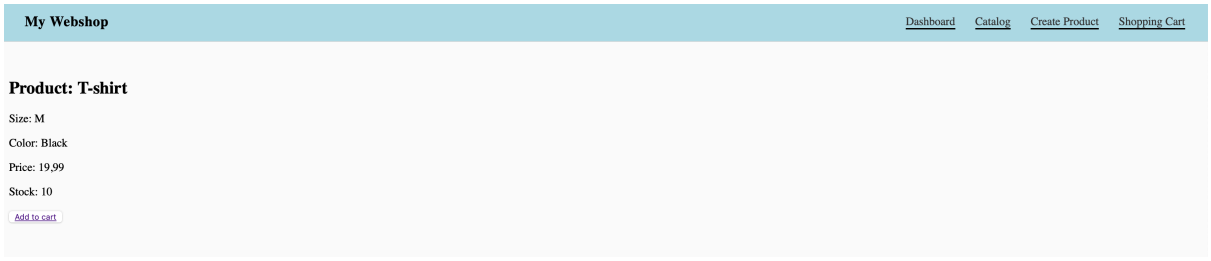
1) The products in the catalog should get a stock / inventory count.

The inventory should be persisted not within our list of products (might exist so far in your ProductService), but separately. The info shall be a hash map containing entries for productId and stock count. Create a

service folder and add an **InventoryService** (acting as a replacement for a data store / Repository) that persists the data how many products are available for each productId. The service needs methods like getStockForProductId(), reduceStockForProductId(). Ensure stock count cannot be lower than zero. Add hardcoded stock values for all products you defined in the ProductService.

2) Enhance the **ProductDetailPage** (Thymeleaf template from the previous exercise) with an info, how many items are available for that product. The Controller needs to fetch the information from the **InventoryService**. The rest of the ProductData information remains fetched from the **ProductService**. In order to get a clean architecture we need a new model class for the ProductDetail screen. It could be named **ProductDetailDTO.java** and combine data from the ProductModel.java, the Integer for the stock information and a utility boolean **isSoldOut**. The Controller providing the get endpoint for the Product Detail screen has now to provide the ProductDetailDTO as a model to the Thymeleaf template. Refactor the template to show model values now not from product but from productDetailDTO: productDetailDTO.product.price, ...

Show also the stock information.



3) Assuming you committed and pushed the code until now, you can now refactor the code. Refactor the ProductDetailController code to not

call the ProductService and the InventoryService separately populating the ProductDetailDTO. The ProductDetailController should only call a class named **ProductDetailFacade** which encapsulates the logic interacting with these two services and also the logic of combining information to populate a DTO. Create a folder facade and add the new facade class. A facade also gets a @Service annotation to allow dependency injection via autowiring in the controller class. Having less dependencies (and less logic to fully populate the DTO) makes the Controller code cleaner and easier to maintain. If the ProductDetailDTO changes any time, we do not need to touch the controller code.

If your controller still has any other calls to the product service, you will still need a dependency on that service, but at least you should be able to delete the dependency on the inventory service.

What is usually done in such a refactoring phase, having introduced a facade is that you add additional methods to the facade, matching the method names of the service and call them like a proxy.

Controller => facade.serviceMethod => service.serviceMethod

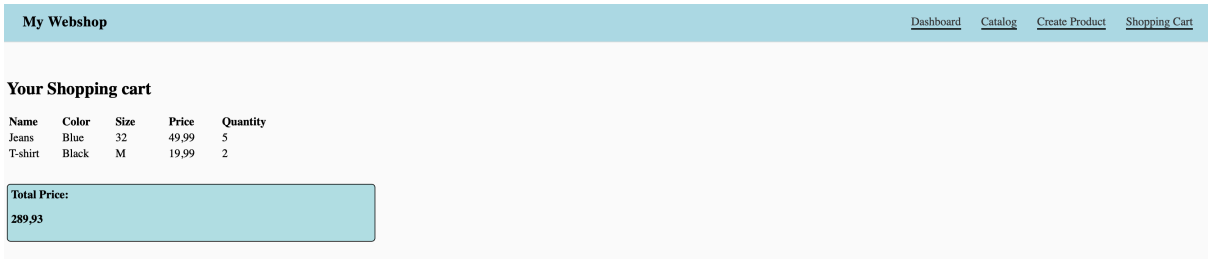
If you do this for all service methods, you can finally delete the direct dependency of the controller to the service.

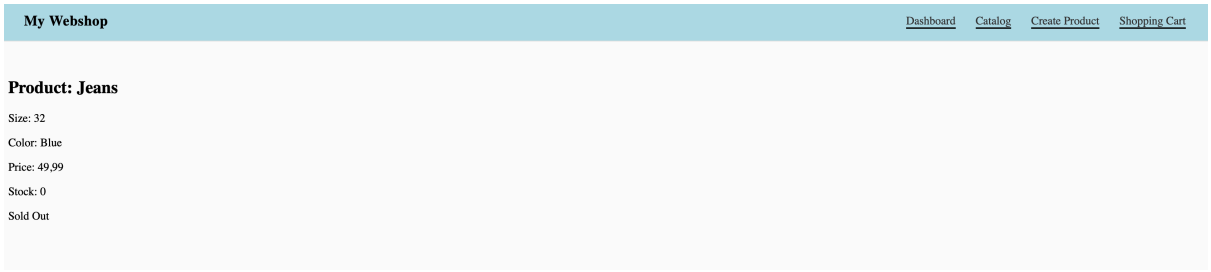
Run your application and make sure the product detail page is still functional.

4) Now add logic to your ShoppingCartController, that every time you add a product to the cart, the stock for this product is updated. Stock is reduced by 1 (calling the InventoryService). If you visit the product detail page again - after having placed the product in your cart - the available stock shall be decreased.

Add a logic with th:if to the Thymeleaf template to highlight „Sold out“ if the stock is zero. Show the „add to cart button“ only if not sold out using th:unless.

Test your application by browsing the catalog, adding products to the cart - until stock is zero.





5) Ensure that you cannot add more products to your cart than existing in the inventory. Hiding a button in the frontend is not secure enough to avoid calling controller endpoints. The **ShoppingCartController** in charge of the „add to cart“ action can also speak with an **AddToCartFacade.java** that talks internally with the **InventoryService** to ensure this condition and only calls the **CartService** to add a product if that is allowed. Refactor the ShoppingCartController to not have direct service dependencies anymore, only a dependency to its AddToCartFacade. The facade has dependencies to CartService and InventoryService.

Write a method addToCart(Long productId) in the facade encapsulating business logic checking for current stock and potentially adding the

product to the cart and reducing the stock. The method shall return the cart to be displayed in the template. No matter if a new product was added or if the cart remains as before.

You might need a proxy method in the facade again for other service calls to fully replace the dependencies in the controller.

Hint: Temporarily remove the condition on the ProductDetail page to hide the „add to cart button“ to test you cannot add sold out products.