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

Exercise 9 - 18.12.2025/12.01.2026

Please do not forget to commit your code and push to GitLab!

Learning goals

Today we want to practice writing **middleware adapters** fixing inconsistencies when connecting components.

We continue coding on our Java Spring Boot webshop application and refresh our knowledge using previously used annotations for dependency injection defining a controller with a POST mapping using a request parameter, as well as creating **facades** and services.

We also want to get to know a problem in dependency injection that enforces using facades. Exercise 3) will cover that.

Practical Exercises

1) We want to practice fixing „**parameter inconsistency**".

Add a „pay now button" to your shopping cart screen. Assume it submits a form with a hidden input field containing the shopping cart total price (in a real world it might contain the id of the shopping cart if that would be persisted in a database but we skipped that part due to complexity).

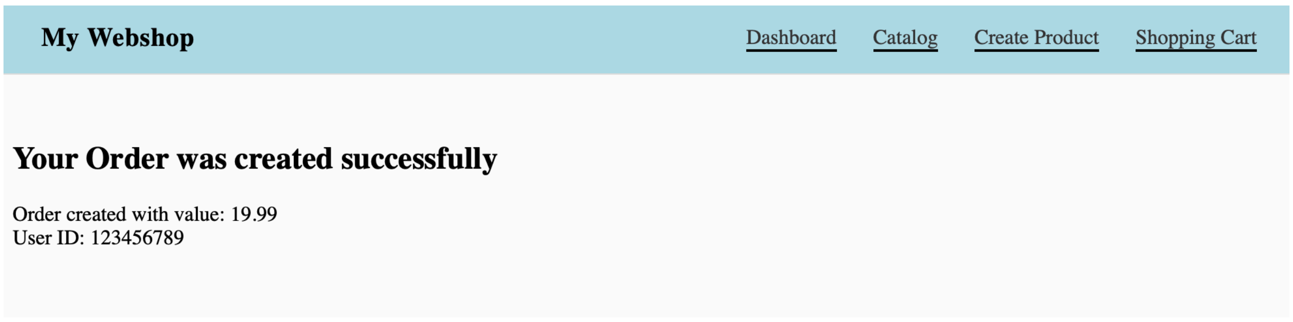
The POST controller endpoint for the **/checkout** action should be located in an **OrderController.java** and creates an order by calling „***finalizeOrder***" in a **OrderService.java**.

The problem we are facing is now, the OrderService method ***finalizeOrder*** which returns an Order might need two arguments: the total price and an userId. This way the order references the user it belongs to.

Modify your existing **UserService.java** class adding a method ***getUserId*** (returning a hardcoded id for now). We do not want a dependency of the UserService in the OrderController. Instead, we can use the Adapter pattern and add a ***OrderFacade.java*** that wires both OrderService and UserService and calls the OrderService method with the fetched userId and the totalPrice. The facade is acting then as our middleware. Create a method ***finalizeOrder*** in the facade which just needs the total price as an argument, so it can be called from the controller. It returns an ***Order*** with a price and the userId. This way, the OrderController no longer has a dependency on the OrderService.

Display the created order with totalPrice and userId in an „order success view" using Thymeleaf templating and a populated model containing needed information.

Solution might look like this:



Your Order was created successfully

Order created with value: 19.99
User ID: 123456789

2) We want to practice fixing „**operation inconsistency**".

Assume to finalize an order, we should also trigger sending a confirmation email. This shall be handled in a new class **EMailService**.
We do not want to trigger a real e-Mail now, but simulate the sending by logging to the console. We can use the **Logger** from java.util.logging.Logger for that. Implement a method ***sendEMail*** which needs the userId as an input param. The methods logs an information that an email has been sent including the userId.

Our **OrderFacade.java** has the method ***finalizeOrder*** which does fetch the userId from **UserService** and calls ***finalizeOrder*** from the **OrderService**. Assume, the OrderService is fixed, we cannot modify its code. Assume now, also the OrderFacade is fixed and we cannot modify it anymore.

Using another adaptor mechanism, we could ensure together with ***OrderFacade.finalizeOrder***, we do call ***EMailService.sendEMail***.

Write an **OrderAdapter.java** (with @Service annotation) that also has a method ***finalizeOrder*** with a parameter for the totalPrice which calls both the OrderFacade.java with ***finalizeOrder*** and the ***EMailService.sendEMail*** method. The OrderAdapter can retrieve the userId from ***OrderFacade.finalizeOrder*** which returns an Order containing the userId. This way, the OrderAdapter has all needed input information to call ***EMailService.sendEMail***.
Change now the evocation in the OrderController to call the **OrderAdapter** instead of the OrderFacade. The OrderAdapter acts now as middleware between the OrderController and the OrderFacade.

3) Be sure your code is committed and pushed until here as you will try out something that you roll back afterwards.

Imagine the OrderService.java would have a dependency on the UserService directly, including a method

```java
public Order finalizeOrderWithTotal(BigDecimal total) {
    String userId = userService.getUserId();
    return new Order(total, userId);
}
```

which would make the **facade** redundant fetching the userId.

Imagine we would also need a method in the OrderService that returns the created order for a user (I used a zero Order for simplicity)

```java
public Order getRecentOrderForUser(String userId) {
    return new Order(BigDecimal.ZERO, userId);
}
```

and the UserService.java knowing the userId has a method

```java
public Order getRecentOrderForUser() {
    String userId = getUserId();
    return this.orderService.getRecentOrderForUser(userId);
}
```

What happens if I autowire the UserService in the OrderService and the OrderService in the UserService?
Try it out by implementing the two methods and run the app. Why does the App fail to start? What is the explanation?

Copy paste the error log to a markdown (ex-9-3-crashlog.md) file in your project.

Roll back the code changes afterwards / comment them out to ensure the app starts again. Push the markdown file to GitLab.
Now you should understand why the facade is not redundant as it decouples the services from each other.