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

Exercise 6 - 02/03.12.2025

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 Java Spring Boot. If you missed previous weeks, please do all the exercises.

We want to get familiar with **paginated requests** using helpers provided by **Spring Data** and the **JPA** package.

We also want to work with the concept of **reusable components** - code units that can be shared rather than copy/pasting. For this it is good practice to **document your interfaces and APIs**. We will use **Javadoc** to read and to provide documentation for used / our code. Assuming we would consume or provide a library package, the internal source code might no longer be accessible. We have to rely on docu then. (We will use Swagger UI as an alternative method for API documentation in a later exercise class).

We will learn about the datatype **BigDecimal** of java.math, which is the industry standard when it comes to price handling. In that context we practice refactoring and create a reusable service component.

Practical Exercises

1) For our product catalog we want to use **pagination** in case the whole catalog / a filtered search result would deliver too many results. This is saving load on our server and database and minimizing data to be downloaded by the browser.
   As our product database should contain 10+ entries, we can limit a catalog page size now to 3 products, just to demonstrate pagination

usage. Assuming the Repository to handle product data is extending JpaRepository you can now create Pageable Requests.

Create a **/catalog-paginated** request mapping to a new catalog template that shows you 3 products per page. Always display 3 products (max) per page.

Hint: In your controller, you can set a default size attribute of 3 like this:

```
@PageableDefault(size = 3) Pageable pageable
```

Add Previous and Next Buttons to the screen that allow to navigate to the next or previous page (you will need th:href).

Hide the next and previous page buttons if there is no page before / after that contains products.

You might need to look into the slides of lecture 5 to get the needed code hints.

Use Page and Pageable of org.springframework.data.domain.

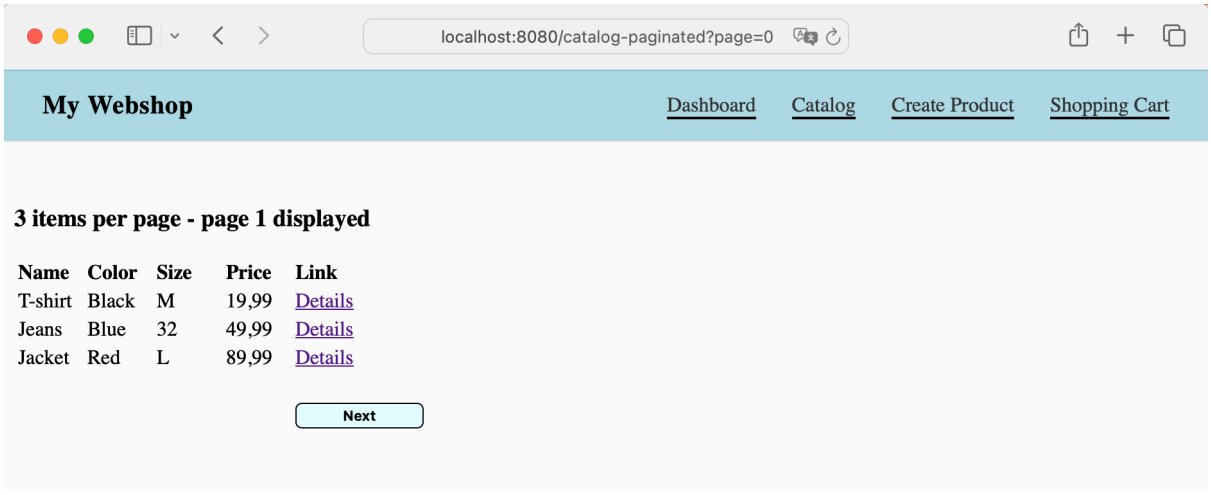
Pageable, Page and Slice are reusable software, the Library package is documented here:

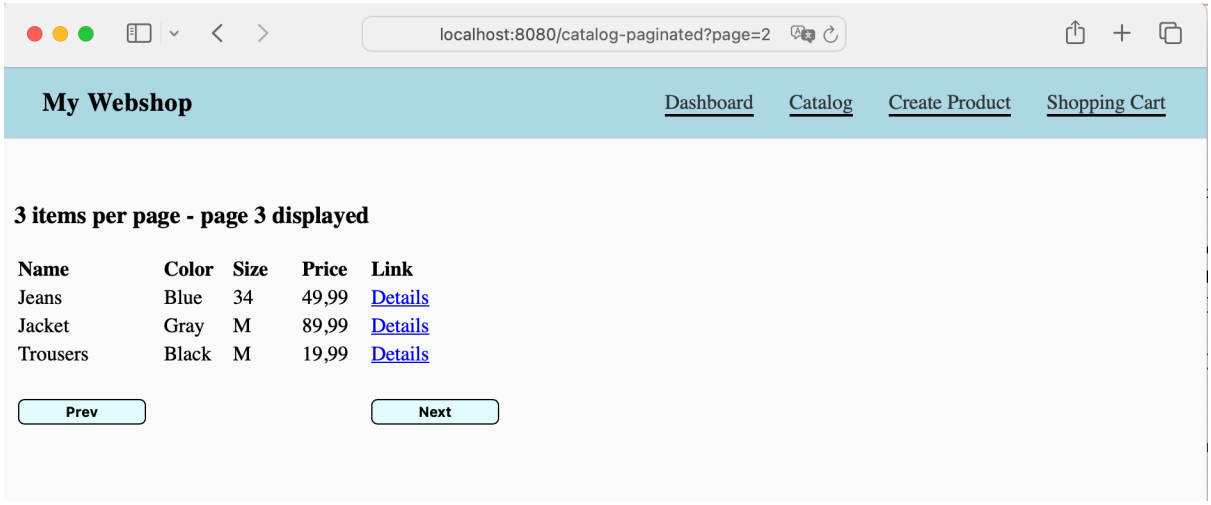
https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/domain/package-summary.html

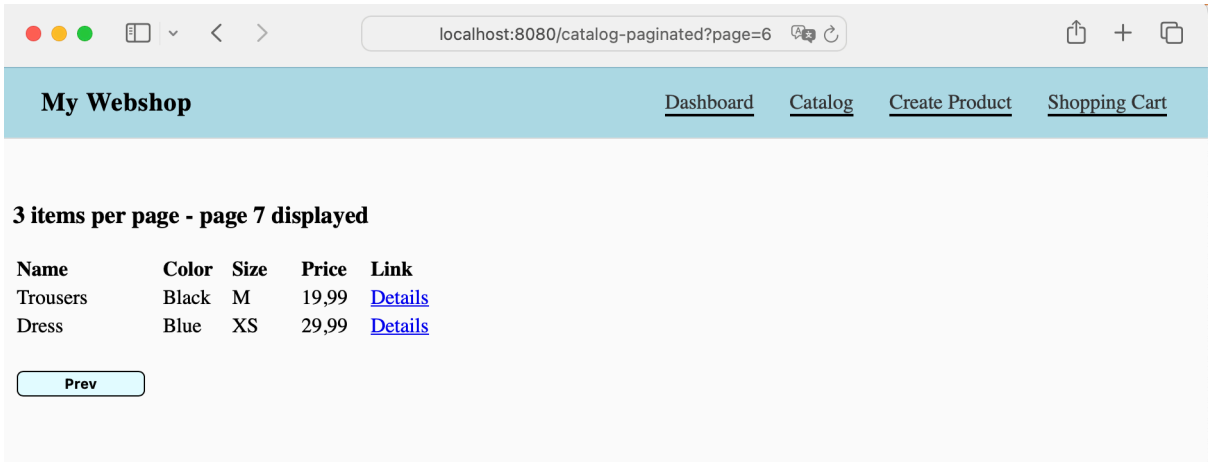
By inspecting the interfaces for Pageable and Page (extends Slice) you can find out methods that help you identifying if your are on the last or first page => populate the model to show buttons conditionally.

You will also find out how to get the list of products from a Page.

The result should look similar to these screens (I have 20 products):





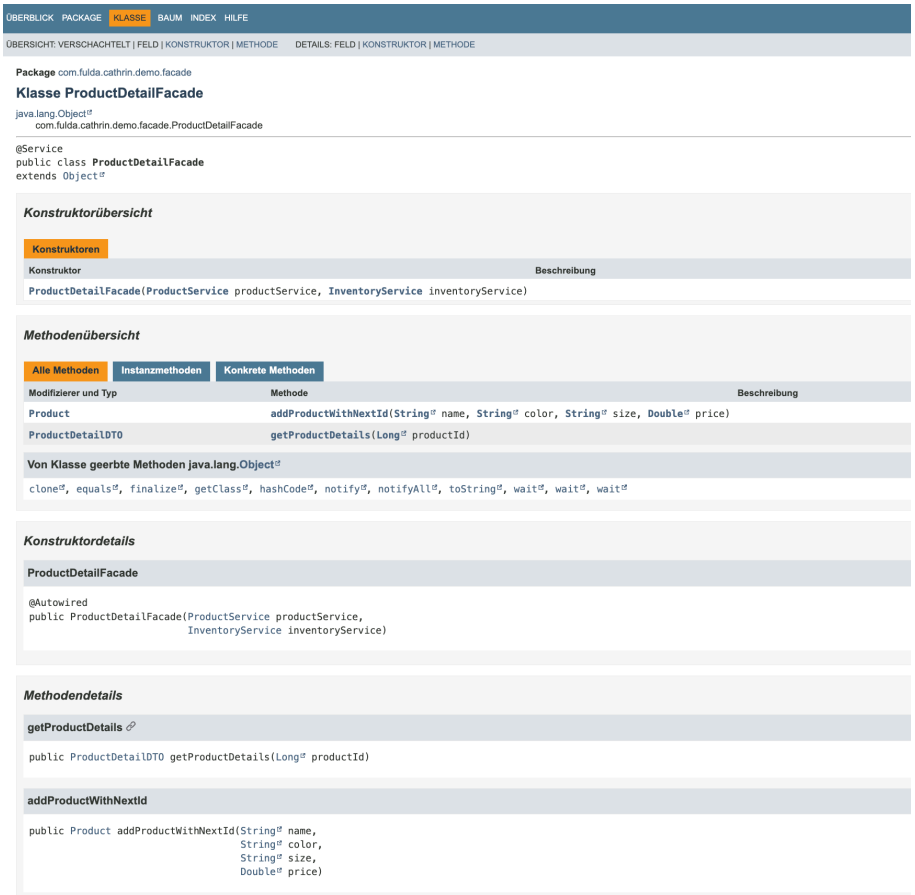


2) In order to share **reusable components** we need **interface documentation**. Create a folder „documentation“ (or java-doc) within your project.

IntelliJ IDEA has a built-in **JavaDoc** generator. Just run „Tools“=>“Generate JavaDoc“, select the documentation folder as output directory and it will create and open HTML files. Ensure that folder is mentioned in .gitignore as it will pollute your repo otherwise.

Alternatively, there is JavaDoc generation provided via a Maven Plugin that you can install or you can use the command line (as it is part of the JDK). Using IDEA might be most convenient.

Generate JavaDoc and browse your code packages (from index.html) to inspect some files, like the ProductDetailFacade. The default result will be sth like this:



ÜBERBLICK PACKAGE KLASSE BAUM INDEX HILFE

ÜBERSICHT: VERSCHACHTELT | FELD | KONSTRUKTOR | METHODE DETAILS: FELD | KONSTRUKTOR | METHODE

Package com.fulda.cathrin.demo.facade

Klasse ProductDetailFacade

java.lang.Object
  com.fulda.cathrin.demo.facade.ProductDetailFacade

```
@Service
public class ProductDetailFacade
extends Object
```

Konstruktorübersicht

Konstruktoren

Konstruktor	Beschreibung
`ProductDetailFacade(ProductService productService, InventoryService inventoryService)`	

Methodenübersicht

Alle Methoden | Instanzmethoden | Konkrete Methoden

Modifizierer und Typ	Methode	Beschreibung
`Product`	`addProductWithNextId(String name, String color, String size, Double price)`	
`ProductDetailDTO`	`getProductDetails(Long productId)`	

Von Klasse geerbte Methoden java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Konstruktordetails

ProductDetailFacade

```
@Autowired
public ProductDetailFacade(ProductService productService,
                           InventoryService inventoryService)
```

Methodendetails

getProductDetails

```
public ProductDetailDTO getProductDetails(Long productId)
```

addProductWithNextId

```
public Product addProductWithNextId(String name,
                                    String color,
                                    String size,
                                    Double price)
```

3) The above documentation might not be **meaningful** enough.

In the default JavaDoc the method summary is showing me the names and input parameters of my method but no description what the method is doing.

You can enhance the documentation for the generated JavaDoc.

You can add **descriptions** to all **public methods or fields** and to a class.

An example is

```
/**
 * The price of the Product is a double - representing the price in Euros
 */
public double getPrice() {
    return price;
}
```

Create an improved documentation for the **Facade** classes annotated with @Service that show what your Facade components **require** (dependencies to be injected => private fields) and what they **provide** (functions with expected parameters - describe the functionality in 1-3 sentences).

Add also **documentation for the product** entity / model class to explain the intention of the object attributes.

Re-generate your JavaDoc.

Reflection (note down yourself or add a Markdown file for answering):

What happens if I add the docu to a private field instead of the public getter method?

What part of our code is not included in the JavaDoc?

4) For correct **pricing**, using a „**BigDecimal**“ datatype instead of a **double** is recommended. It is a secure type for using monetary values as it calculates exact values. You cannot rely on double values to be exact - even with subtraction rounding issues can happen! Refactor your product model / entity class to use BigDecimal for the price. Make sure the datatype is changed everywhere. Code will compile if you changed all places. (Database remains intact!)

Hint: IDEA => right click „Refactor“=>“Type Migration“ will help you.

Hint: you might need to change manually your LoadProductDatabase to initialize your products with prices:

```
new BigDecimal("19.99")
```

Further docu (methods like .multiply() and .add()) is provided here:

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/math/BigDecimal.html

Change all your price fields from double to BigDecimal. You will also need to update the object and calculations for the ShoppingCart to calculate the totalPrice in BigDecimal.

From which java package do we import this type?

5) We want to create **components which are reusable** - if they are intended to be used more than once (in future).

Imagine a webshop needs to deal with prices a lot. E.g. we need to ensure prices are rounded to cents / 2 digits after the comma.

Write a reusable @Service **PriceCalculationService** that offers a method to

- round prices

A hidden implementation detail will be that it rounds up from 0.5.

Hint: use import java.math.RoundingMode

Use that service to avoid (inconsistent) rounding logic in different parts of your code. Replace rounding code logic by calling the PriceCalculationService.

It might be for now that this service is only used once to calculate the total price of your shopping cart. In future we will need it to round prices after currency conversion, voucher application, ...

Create also some JavaDoc for the public Service method.