Read me- Alexia\_Sandor

I organized the project using a typical Spring Boot architecture, with a clear separation into layers: controllers for API exposure logic, services for business logic, DTOs for data transfer, entities for the core model, repositories for database access, and CSV files in the resources folder for data loading.

**Instructions on how to build and run the application.**

**Requirements:**

Java 17+

- Maven

- PostgreSQL

**Steps:**

-> Clone the repository

-> Update application.properties with your PostgreSQL credentials

-> Run the application

-> Testing can be done in Postman by adding the appropriate IDs where required. In the project folder, you will find - test examples

<http://localhost:8080/basket/create>

<http://localhost:8080/basket/optimize/752>

<http://localhost:8080/discount/bestDiscounts>

<http://localhost:8080/discount/insertDiscount>

<http://localhost:8080/discount/newlyAdded>

<http://localhost:8080/product/insertProduct>

<http://localhost:8080/product/historyGraphs>

<http://localhost:8080/product/pricePerUnit>

<http://localhost:8080/priceAlert/createPriceAlert>

<http://localhost:8080/user/insert>

<http://localhost:8080/product/updatePrice>

<http://localhost:8080/user/getMessage/752>

**Any assumptions made or simplifications**

The dates in the CSV filenames are treated as the date from which the product or discount becomes available

No authentication or authorization is implemented for users.

Discounts are applied based on defined rules (the most recent active discount).

Messages to users are stored in a field within the user entity, not in a dedicated notification table.

**Database Storage**

To build this project, I decided to save the .csv files into a PostgreSQL database using a DataLoaderService, which loads data into the appropriate table (Product or Discount). The CSV filenames, being in the format kaufland\_2025-05-01.csv, were treated as the date when the products become available in that store.

When saving discounts, I added a new field representing the date and time the discount was created (used to solve Task 3). For existing discounts that were already present in the CSV files and imported into the database, I generated a random creation time.

**New Functionalities Added (in addition to the required ones)**

I implemented basket creation, basket search by ID, and displayed both the original and optimized basket contents along with the total price calculation in both cases. I also implemented discount creation, new product creation, alert creation, user creation, product price update, and user message retrieval.

**Explanation of How Tasks Were Implemented and the Thinking Behind Them**

1. ***Daily Shopping Basket Monitoring***  
   -> Discounts are applied to a basket based on the period we’re in. After applying discounts, the best price-to-quantity option is selected. The return consists of a list of lists (depending on stores), and the products in the basket are replaced in the database with cheaper alternatives. In a real application, we would use the actual current date, but because I needed to test products/discounts from past dates (and the project was developed between 14.05 and 17.05), I added a new field that receives a hypothetical current date (sent in the request) and identifies the current period, so that the available discounts are applied correctly.
2. ***Best Discounts***For this task, I wrote a function that returns the best available discounts valid during the current period. For example, if we are on 2025-05-05, we receive a list of all discounts valid for that date. If a discount does not cover this period, it will not be displayed.
3. ***New Discounts***For this task, I filtered discounts created in the last 24 hours. I added a new field called creationDay to the Discount entity, of type LocalDateTime, so we have access to both the date and time when the discount was created***.***
4. ***Dynamic Price History Graphs***This task involves combinations of filters. To avoid creating multiple endpoints for every possible filter combination, I used the Specification<T> interface. To test this task, you must first add multiple products. I've included the necessary test cases in the folder: "examples of testing".
5. ***Product Substitutes & Recommendations***This task does not require additional explanation.
6. ***Custom Price Alert***For this task, I added two new entities: User and PriceAlertService. A user can set alerts for multiple products, so there is a one-to-many relationship. When a new alert is created and a product’s price is updated (using the update method), we check whether that product is associated with any alerts. If it is, and the new price is less than or equal to the targetPrice, a message is sent to the user who set the alert. This message is saved in the database.