Date	# Hours	Description of work
25.03.2025	3	I implemented full cart functionality for authenticated users. First, I created the `Cart` and `CartItem` entity classes. Each cart is linked one-to-one with a user, and a cart can contain multiple items. Each `CartItem` stores information about the product and its quantity. I added a cart field to the `User` entity and updated the `UserService` so that when a user registers, a new cart is automatically created and linked to them. Next, I created `CartRepository` and `CartItemRepository` to manage the carts and their items in the database. In the `CartController`, I added a `/cart` endpoint to load the user's cart and display it using Thymeleaf. Additionally, I implemented a `/cart/add` POST endpoint to handle adding products to the cart. This endpoint checks if the item already exists; if it does, it updates the quantity; if not, it creates a new cart item. I also added the necessary getters and setters for quantity in the `CartItem` class. On the frontend, I updated the `product-details.html` page to include a form that allows users to select a quantity and submit their choice. The cart page displays all cart items in a table, showing the product name, description, price, quantity, and total. It also manages scenarios where the cart is empty.
26.03.2025	2.5	In the CartController, I replaced the old `addToCart` method, which used form parameters, with a new version that accepts a `CartItemRequest` object in JSON format. This change facilitates smoother interactions on the frontend without requiring full-page reloads. I also implemented proper HTTP responses using `ResponseEntity` to return appropriate status codes and messages. To support this functionality, I created a new Data Transfer Object (DTO) class called `CartItemRequest`, which contains fields for `productId` and `quantity`. Additionally, I added the missing getter and setter methods in the `CartItem` entity for both the cart and product. On the frontend, I updated the product details page to handle cart addition using JavaScript's `fetch` method, sending data as JSON and displaying a Bootstrap toast notification to indicate success or error. Furthermore, I developed a `cart-checkout.html` page styled consistently with the rest of the app, which displays products added to the cart, including their name, price, quantity, and total. I added a reusable navbar and footer across some pages to create a more polished and cohesive user experience. I improved the cart functionality by implementing a complete checkout flow. In the CartController, I added a `/cart/summary` endpoint to show the cart summary with total quantity and price, along with a `cart-summary.html` view for a clean presentation. The Cart entity now features a `List<CartItem>` with appropriate JPA annotations, enabling access to all items in the user's cart. I also created

		endpoints for updating item quantities and removing items at `/cart/updateQuantity` and `/cart/removeItem`. On the frontend, I developed a styled `cart.html` page listing all items with forms for updates and removals, linked to a checkout summary via a "Checkout" button. The `cart-checkout.html` page collects shipping and payment information using a structured form while maintaining a consistent navbar and footer across all views. In the CartController, I added GET and POST handlers for the `/cart/checkout` route. The `getCheckout()` method returns the `cart-checkout.html` view, and the `completeCheckout()` method processes the form submission, capturing shipping and payment details after verifying user authentication. I saved a new Transaction post-purchase by building a Transaction object with shipping details and purchase information. After marking the status as "SUCCESS" and saving to the database, I passed the `transactionId` and `totalPrice` to the success page. Additionally, I updated entity relationships by linking Product to transactions with a `@ManyToMany` association and created a join table called `transaction_product`. The Transaction entity now supports multiple products, laying the groundwork for future multi-item checkouts.
27.03.2025	2	I implemented a basic product recommendation system and developed the data seeding infrastructure to support it. I created a `.env` file and utilized the `dotenv` package to securely load the database credentials into the application. In the `requirements.txt` file, I listed the necessary libraries, including `mysql-connector-python`, `pandas`, `SQLAlchemy`, and `scikit-learn`. I added `alchemy.py` to manage SQLAlchemy-based connections and to facilitate table reflection for easier Object-Relational Mapping (ORM) access to existing MySQL tables. Additionally, I created `basic.py` to implement the recommendation logic, which uses cosine similarity based on product categories and normalized prices, storing the results in a new recommendations table. Furthermore, I developed `seed.py` to generate realistic fake data using the `Faker` library for users, entrepreneur profiles, products, and thousands of transactions. This script ensures relational integrity, generates consistent random data, and populates the MySQL database cleanly through `cursor.execute()` and `commit()`. This setup lays the groundwork for building personalized recommendation features within the application. I implemented a system to display personalized product recommendations directly on the product details page. To achieve this, I created a Recommendation entity to define the relationships between a product and its recommended products. This involved using two

		relationships: one for the main product and another for the recommended product. In the Product entity, I added a relationship to connect each product with its list of recommendations. On the frontend, I updated the product details page to include a new "Recommendations" section located below the main product information. This section loops through the recommendations using product.getRecommendations() and displays each recommended product as a card, showcasing its image, name, category, and a link to its detail page. There is also more advanced file for recomedations but it reqires a lot of purchase history in transactions table. I will stick to basic recomendations but category and the price.
30.03.2025	2	The python script wasn't working, and the random generation in the database user and products. I have no idea what happened, had to reboot computer few times and created a list of comands that made everything work: cd desktop cd W25_4495_S2_ValeriiaN cd reccomendations Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Scope CurrentUser venv/Scripts/activate pip install -r requirements cd src python basic.py python -m db.seed python basic.py Writing progress report.

I implemented full cart functionality for authenticated users by creating `Cart` and `CartItem` entity classes. Each cart links one-to-one with a user and can hold multiple items. A new cart is automatically created for users upon registration.

I developed `CartRepository` and `CartItemRepository` to manage cart data in the database and added a `/cart` endpoint in `CartController` to display the user's cart using Thymeleaf. The `/cart/add` POST endpoint handles adding products to the cart, either updating quantities or creating new items.

On the frontend, I modified the `product-details.html` page to allow users to select quantities via a form and updated the cart display to show item details. I also improved the `addToCart` method to accept a JSON `CartItemRequest` object, and included appropriate HTTP responses with `ResponseEntity`.

To enhance user experience, I added a `cart-checkout.html` page and a reusable navbar and footer. I implemented a complete checkout flow with a `/cart/summary` endpoint for displaying cart totals, and developed pages for updating item quantities and removing items.

The `cart-checkout.html` page collects shipping and payment information. The CartController includes new GET and POST handlers for `/cart/checkout`, and upon successful checkout, I created a Transaction object storing relevant details.

`basic.py` implements a recommendation logic using cosine similarity. Finally, I developed `seed.py` to generate realistic fake data while ensuring relational integrity.

In the upcoming weeks, I will focus on polishing the design. Right now, I have completed the full lifecycle of the product purchase, as I promised, but I still need to finalize the design and validations. I won't be making any changes to the functionality because I'm concerned that something might break, especially since I've recently merged some features. I encountered several issues that caused the entire program to halt, and I’m not entirely sure how I resolved them. I basically used a trial-and-error approach. For example, the database stopped responding, and I didn’t even include the hours I spent on that in my progress report. I panicked for a few hours but eventually deleted XAMPP and downloaded MySQL Workbench, which resolved the issue.