EECS 4413 F25 Team Project Report

Team Composition

- Team name: PMN
- Team member 1: Priya Gill (219407329)
- Team member 2: Maryam Ahmadi Tabatabaei (218058750)
- Team member 3: Natalie Lewis (219360320)

The team collaborated to split up the tasks based on grouping the functional requirements by the following: Catalog, Customer Accounts, Cart, Order, and Admin. This is more beneficial than a front end and back end split as using MVC can make things complicated when separating the two. Initially, we designed the database as this would ensure that group members can work on it simultaneously. We then populated the database with sample information, so that the rest of the application can be built surrounding this. Initially, we had an additional group member, Aya, who created the core catalog page. Unfortunately, Aya dropped the course.

Priya:

I initially set up a Github repository by initializing a Spring Boot backend, creating the initial Next.js frontend, and connecting the backend to a local MySQL database. In the beginning we helped design the database by creating entity files, and once the database schema was finalized we moved onto developing features. The first feature I implemented was the Cart component, which included adding products from the catalog to the cart, maintaining the cart through session tracking, and supporting in-cart actions such as updating item quantities and removing items. Next, I worked on the product view component which showcases the description of the product, allows customers to select a size, and updates price accordingly. I also created the Admin inventory view to visualize all of the products and product variants, and then allow admin users to update stock quantities and add new products or variants. Near the end, I focused on deploying the application using Docker, which was challenging since it was my first time using it. After lots of research and experimentation, I was able to run the app using Docker Compose and Docker Hub.

Maryam:

In the beginning stages, I combed through the document to come up with an action plan to split up the work. I took all the requirements and split them up into related groups, so that we can split up the tasks, mimicking user stories and tickets. Myself and Priya, took a large part in designing the database schema. We deemed this to be the foundation of our project before we continued on to coding the rest of the project. Also, I seeded the database with sample products, accounts, orders, etc. I implemented the checkout page, this included the retrieval of the user information, the cart information, the form, then created the order objects, reduced the inventory in the backend and outputted an order confirmation page. To implement the checkout and orders, I needed to understand all aspects of the application, like the cart and the users. Additionally, I added payment and address information to the customer. I implemented the Admin pages for Sales History and Order Management. Initially, SpringBoot was a bit of a learning curve for me, but by the end, I found it very simple to use.

Natalie:

My role on the team was more front-end-focused given my lack of experience and understanding of database systems. Therefore, I was tasked with designing and implementing the signup, login, logout, account page, and past orders page. However, all of these pages were involved with getting data or updating information on the backend, so I had to understand how the data was configured with RESTful APIs and how to retrieve it by analyzing the JSON objects. I also handled validation on the backend when users signup, login, and logout. I implemented an authentication service, control, and model in order to add new customers to the backend and update current users when they make changes to their account, including the admin view. Even though I didn't do much work on the backend, I still got to learn about its functionality and the different APIs used. I learned the architecture, how to approve authentication, and tested the backend features through the frontend integration with the help of the other members.

We attest that the contributions described above are accurate and reflect everyone's individual work and involvement in this team project.

Priya Gill

Maryam Ahmadi Tabatabaei

Natalie Lewis

Date: December 22, 2025

Table of Contents

Introduction

The world of skincare can be difficult to navigate. Aura Beauty is a one-stop shop for all things skincare. We created a Business to Consumer (B2C) website where users can shop for skincare products tailored to their specific concerns. We know that finding a skincare routine is a very personal journey so we wanted customers to be able to find products suited for them. For instance, one may be looking for a moisturizer for their dry skin, but they may not realize that a serum might also cater to those needs. Aura Beauty serves to connect users to products and simplify the skincare world. We modelled this website to take inspiration from other skincare ecommerce websites, but also added some of our own innovative features.

Our team implemented a RESTful backend using Spring Boot, where the app exposes REST APIs that communicate with the frontend using JSON for the request and response. To support this, we utilized the Data Access Object (DAO) pattern, using Spring Boot's JPA interfaces in our repository folder. We also applied the Model-View-Controller (MVC) pattern with Spring Boot's backend configuration by building models to represent the main entities and using controllers to handle the HTTP requests and responses, as well as communication between the frontend and the database. To continue the separation of the API layer, we used Data Transfer Objects to control the data exchanged between the backend and the frontend.

The strengths of our design and implementation are our clean architecture and use of the DAO pattern. This structure separates the business logic from the database and presentation logic by using the following request structure: HTTP request → Controller → Service → Repository → Database. One weakness of our design is that there is some repetition because each functionality has to be passed through many layers. One weakness of our implementation is that there is limited error and exception handling.

Architecture description

At a high-level, the architecture pattern we used for our project is the 3-tier architecture since we have a clear separation of layers such as the presentation layer, the application layer and the data layer. The presentation layer consists of the view component, and this is what the user directly interacts with. The application layer contains the backend Spring Boot components, handling all communication between the frontend and the database. The data layer consists of our MySQL database instance, which stores all application data. This architecture ensures that the user never interacts directly with the database, and that all the business logic and data access are handled securely by the application layer.

Within the application layer, we also follow the MVC architecture pattern, as shown in Figure 1. This pattern separates our Spring Boot backend into controllers, services, repositories, and models. This helps organize the business logic, manage data flow, and maintain a clear separation of responsibilities.

The controllers handle incoming HTTP requests, validate inputs, and call the appropriate service methods. They are built on the RESTful API standard, and they also enable our backend to provide web services for the application.

The services contain all of the business logic, they process the data, and communicate between controllers and repositories.

The repositories are interfaces that follow the DAO (Data Access Object) design pattern and manage database access by performing CRUD operations on model entities. They typically extend JpaRepository from the Spring Framework to help simplify database interactions.

The model classes represent the application's business entities that are passed between application layers, and they essentially define the data structures used throughout the backend. We implemented these model classes as JPA entities so that we could map them directly to database tables.

We also have a few DTO (Data Transfer Object) classes in our application layer, which are designed to combine properties from multiple tables. This makes it easier to send only the necessary data to the frontend for easier use and better performance.

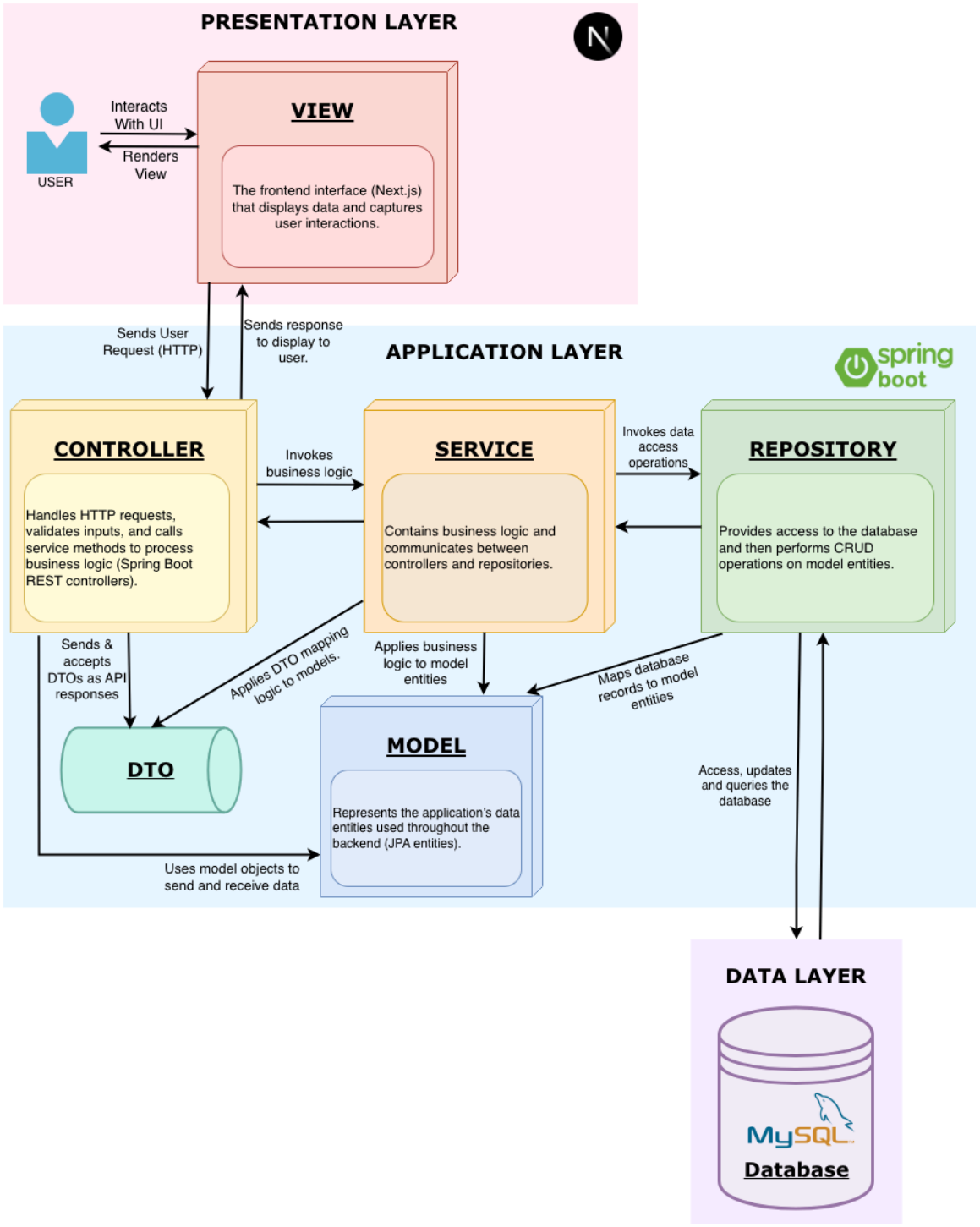

Figure 1: The architecture diagram showcasing 3-tier and MVC Architecture Patterns

The sequence diagram in Figure 2, shows the end-to-end order flow for our e-commerce site, where a customer browses products, adds items to a cart, and proceeds to checkout. It ensures the user is authenticated before placing an order and ends with order processing and confirmation.

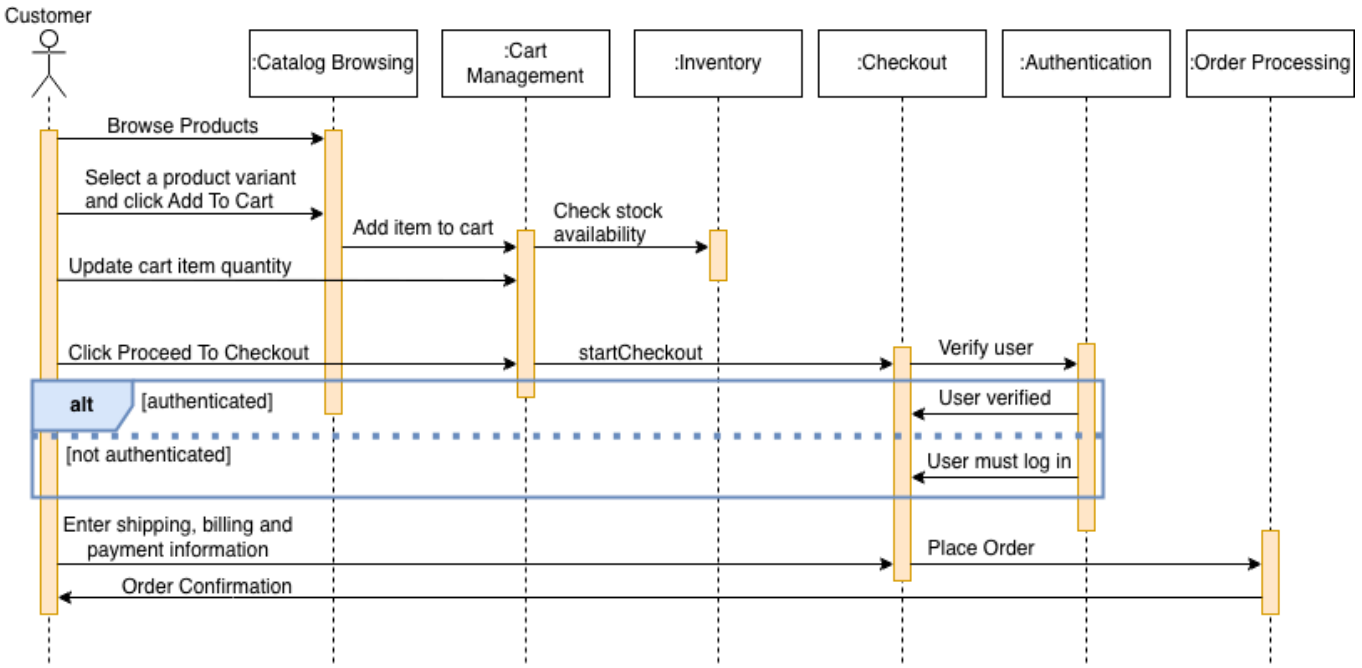

Figure 2: Sequence Diagram for a Customer Placing An Order

Our system is designed for two different actors, Customers and Administrators. The customers use our system to place orders, manage their accounts, and browse, while the Administrators use it to manage their inventory, sales, and customer accounts. The use case diagrams for Customers are shown in Figure 3, and the use case diagram for Administrators is shown in Figure 4.

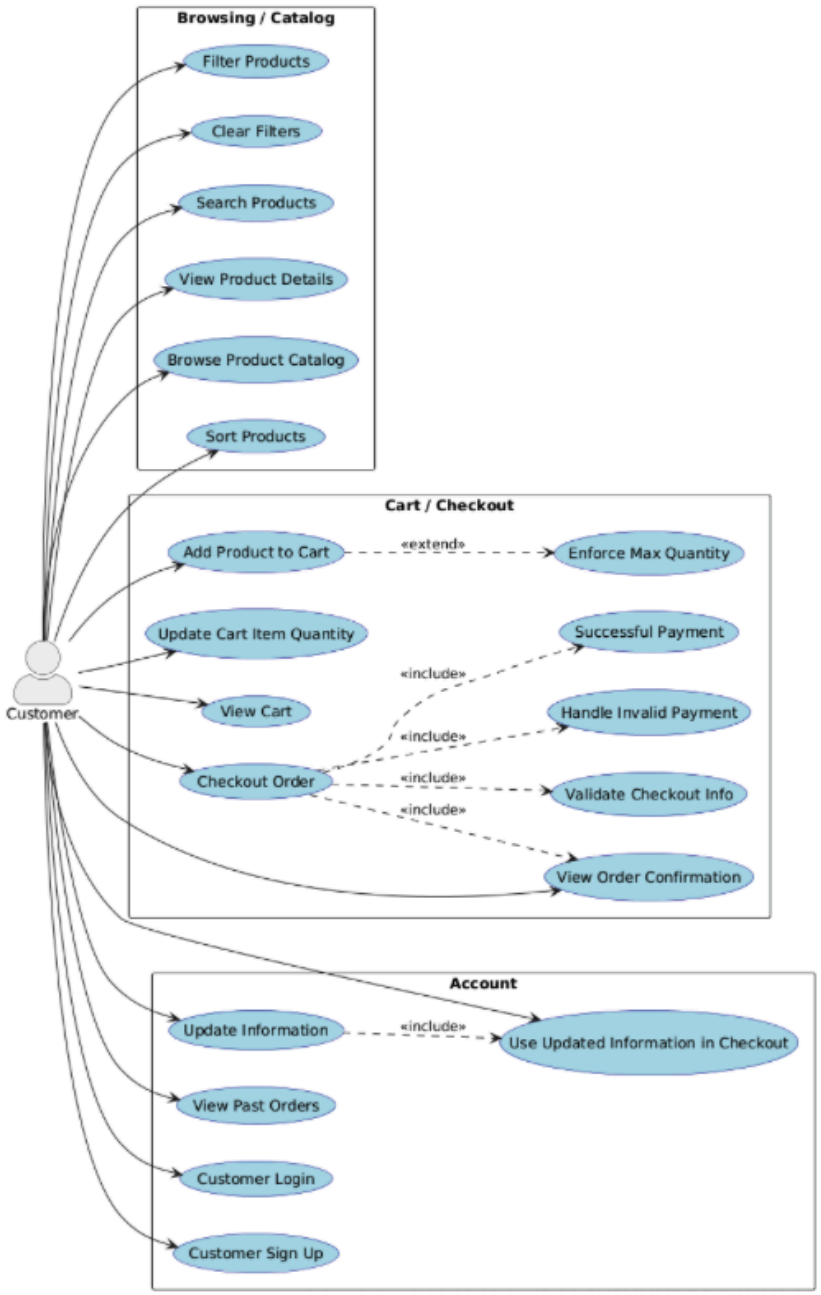

Figure 3: Use Case Diagram for Customers

Figure 4: Use Case Diagram for Administrators

Design Description

Design Patterns

Our project uses the MVC architecture as outlined in the previous section. The model holds our entity classes, the views are in the front-end, while the controllers use REST. The DAO is implemented through our repository files that directly query and update the SQL database. To implement best practices, we added a service layer between the controller and repository, thus implementing a Proxy Design Pattern. The services handle the business logic and act as a proxy before the changes are sent to the repository. The Observer pattern was used for the Cart to be able to update the cart items, number of items in the cart and UI every time changes were made. There is a cart context under the components folder in the front end that would act as the subscriber, while the Cart Controller implements the changes and acts as the Publisher. Additionally, for Order, we implemented the Composite Design Pattern, as Orders are made up of a composite of OrderItems, among other things like Address and Customer. The Composite Pattern also holds for Cart and CartItems.

Class Diagram

For the sake of simplicity, we include the model layer with its attributes, the service layer with its methods, and a skeleton for the repository layer just to show logic. We did not want to show a cluttered UML diagram so we left out the controllers, and information within the repository layer. See Figure 5 below for the UML Diagram or visit this link for a clear view.

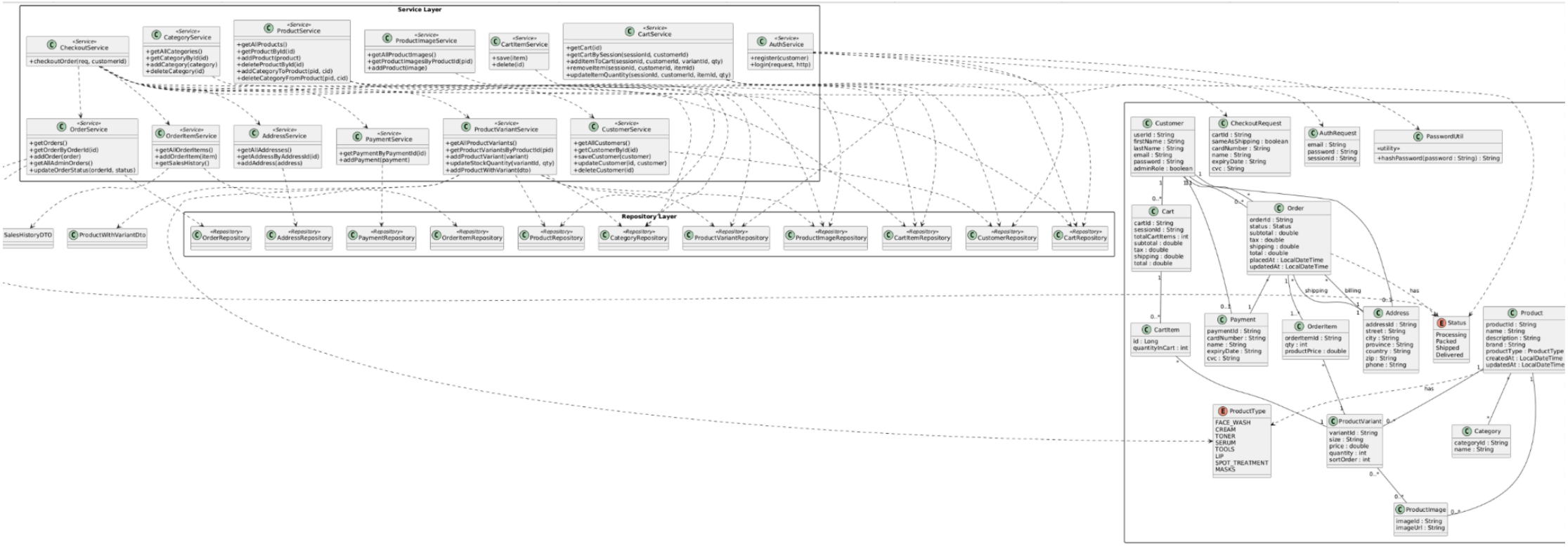

Figure 5: UML Class Diagram

Database Schema

We designed our database to follow principles of database design by ensuring there is no redundancy, and is thus normalized, while adhering to our functionality in the simplest way. The database schema can be viewed in Figure 6. To keep track of all accounts in the system, we maintained a customer table that included foreign keys to address and payment tables. We added the attribute “admin_role” to keep track of the one admin account in our system. The customer table held all login information like email and password, as well as the users information like their first name and last name.

We needed to store the products of our store (Product table), however we also had different product variants because there were different sizes of these products; for example travel sized, full sized, value sized, all in varying volumes depending on the item. To reduce redundancy, we implemented a product variant table that had a foreign key to the product table. The products have multiple images associated with them, and some variants might have their own images of that particular size of product. Additionally products are of varying categories, and can belong to more than one category.

For the cart functionality, we wanted to be able to store the cart information so that if a user logs out, their cart is saved into the cart table. When they log back in, the cart is retrieved. To do this the cart table holds information of the customer, subtotal, shipping fee, total etc. The cart_items table has a foreign key to product variants and the cart it belongs to.

To maintain information on orders, we followed a similar structure to that of cart, by separating orders table and order item tables to reduce redundancy. Order entries also included information on the shipping and billing addresses, payment information, the order status, the time the order was placed, and the time at which the order was updated.

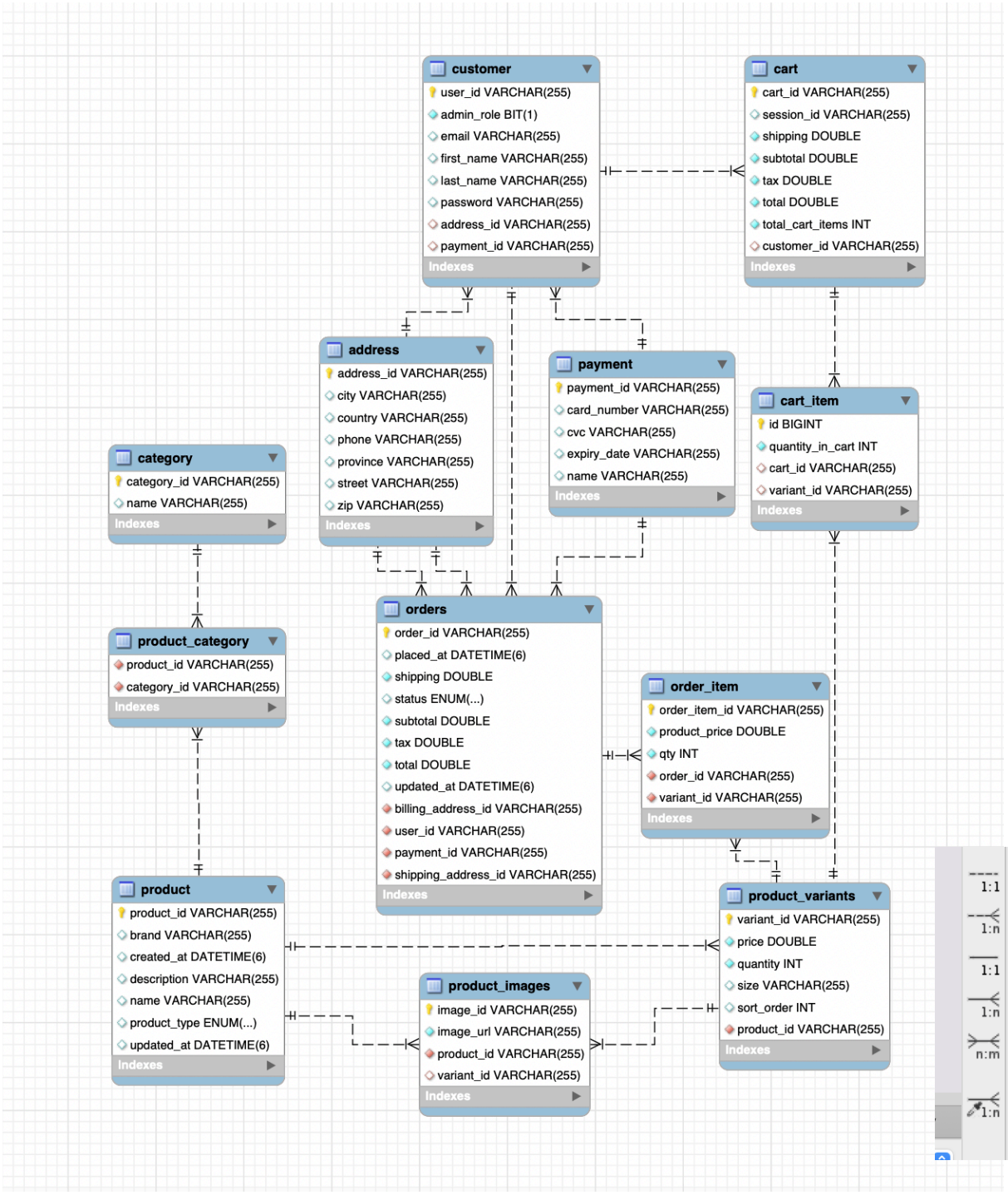

Figure 6: Database Schema E/R Diagram

Advanced Features (Beyond the requirements)

For our e-commerce website, we tried our best to include some advanced and innovative features that went beyond the project requirements.

Categories

Starting with the catalog page, we decided that it would be a good idea for our products to have categories. This is because our retail site is selling products related to skincare and selfcare, so it is easy to categorize these products in regards to skin concerns or targets that customers may have. Some examples of the categories we have are hydrating, oily skin, exfoliating, etc. On our catalog page, an innovative feature we added in regards to these categories is that users are able to filter the products by selecting 1 to many categories, as shown in Figure 7 , so that they can find the products that target any of their concerns. We also list the categories in the product view under "Suitable For", as shown in Figure 8 so that they are clearly shown to the customer.

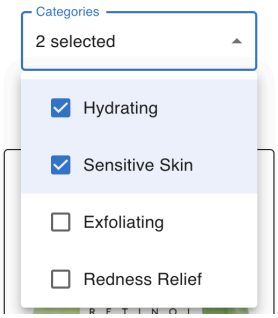

Figure 7: shows categories users can filter by.

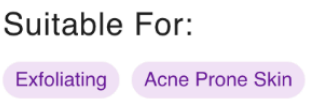

Figure 8: displays the tags of categories relating to a face wash.

Image Carousel

In the product view, we also added an image carousel component to display multiple images for the products rather than only showing 1 image. This allows us to have more flexibility when it comes to adding pictures for our products since we are not limited to any number constraint. Please see Figure 9 below.

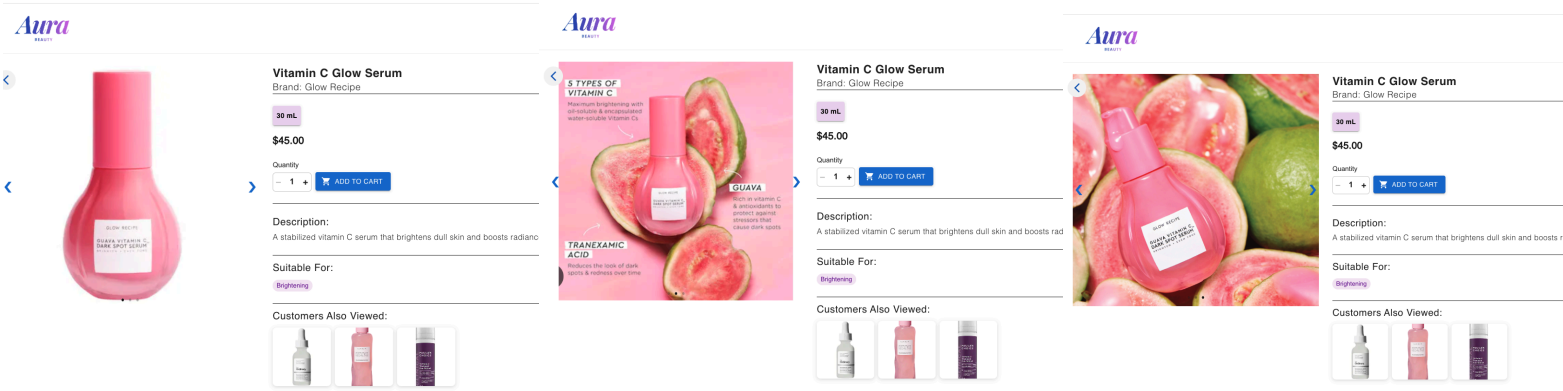

Figure 9: shows the image carousel for a Vitamin C Serum.

Product Inventory Handling

We ensure that users are informed when items are sold out and that they cannot add any more of that item. Additionally, if the user tries to add more than the quantity in stock, a warning message is displayed to them (as shown in Figure 10) and the backend prevents items from being added to cart. Figure 11 shows that when an item is sold out the user may not add it. Also, it will not let a user add more quantities of a product than what is available in the cart, as shown in Figure 12.

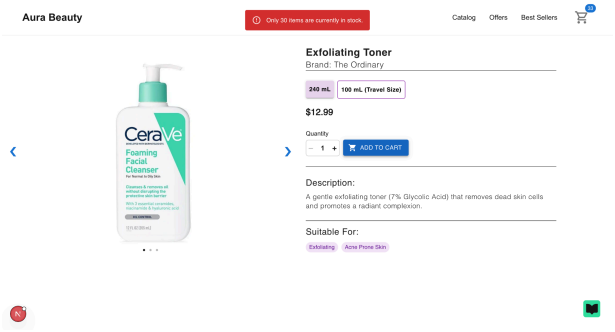

Figure 10: Warning message when trying to add more than stock quantity.

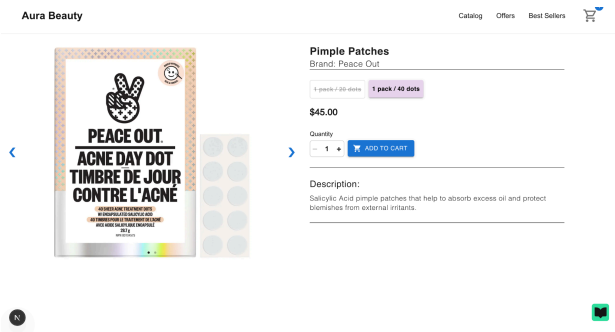

Figure 11: a sold out size of an item.

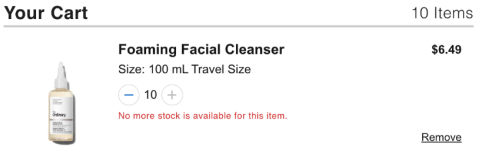

Figure 12: Trying to add more than quantity in stock.

Recommended Products

To improve the user experience, we added a recommended products section, as shown in Figure 13. This section includes three recommended products based on the category of the product the user is viewing. The logic behind this is that if a customer is looking for a hydrating moisturiser, they may also be interested in hydrating masks, and serums.

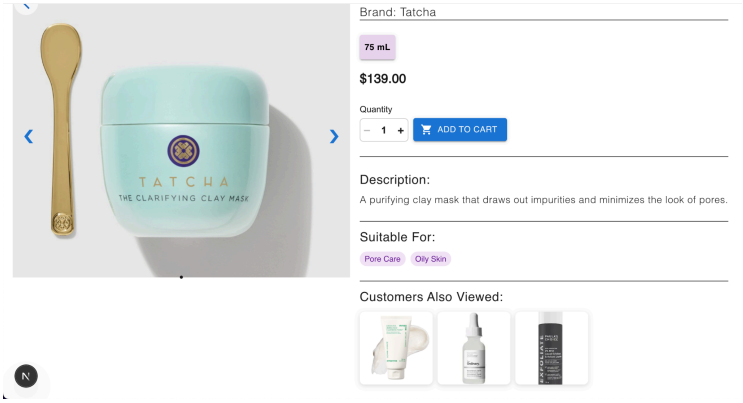

Figure 13: recommended products shown.

Password Hashing & Security

Although not initially outlined in the requirements we decided to take the extra step to hash the passwords of user accounts. This adds an additional layer of security to the user's information.

Cart Saved for User

We decided to handle the Cart for guest users and account holders differently, so that we can ensure the cart is saved for when a user logs out and back in. If the user is a guest, the regular session tracking with 30 min expiry is used to remember their cart. However, if the user is logged in, we assign the user info to the cart so that if the user signs in again later, all their history is saved.

Admin Innovative Tasks

For any ecommerce website, the store managers would need to keep track of fulfilling orders. For this reason we decided it was crucial to add an order management feature to our Admin functionalities. In this view, the admin can not only track and filter through different orders, but they can also change the status of the orders once they've been packed or sent out for delivery, etc. These changes will be visible in the Order Summary page. Please see Figure 14 below.

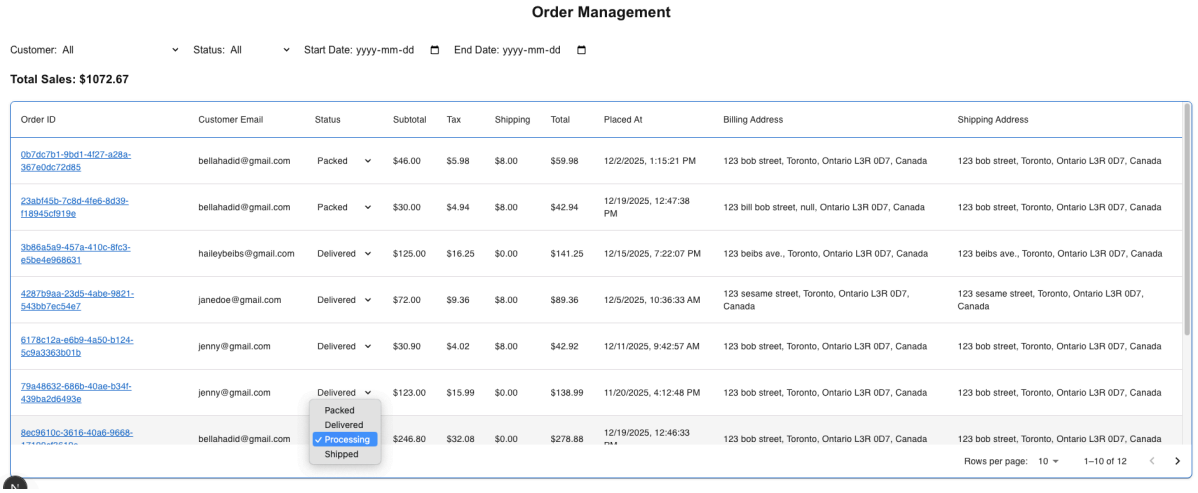

Figure 14: Displays the Order Management view for Admin, and the status change functionality.

Web Services

Our Spring Boot controllers expose multiple REST API web services, which include services for customers, products, orders, payment, cart, etc. To document all of our REST API endpoints, we have provided a Swagger-based API documentation, available at: http://localhost:8080/swagger-ui/index.html .

Implementation

The technologies we considered and decided to use are Next.js, HTML, CSS for the frontend, and Java, MySQL, and Spring Boot, specifically Spring Web and Spring Data JPA. This was our final decision because all team members were familiar with the technologies, except Spring Boot. Most of our group did not have prior experience using Spring Boot and Next.js, which ended up being a learning curve. In the beginning stages the group had to collaborate to overcome this; then, when everyone got more comfortable with the new technologies, the group branched off to do individual work. In comparison to the Java Servlet and JSP pages covered in class, Servlets and JSPs tend to mix business logic with the presentation logic. This makes it more difficult to maintain applications, especially as the data increases. However, Spring Boot provides built-in support for RESTful API development and database integration. Considering some group members were not as comfortable with SQL operations, using Spring Boot repositories allowed us to use already implemented methods (provided by JpaRepository). This creates a clean layered architecture with controllers, services, and repositories, making it easier to understand, maintain, and extend. By choosing Spring Boot, we were able to focus more on meeting the application requirements rather than the framework configuration. Compared to Servlet/JSP, Spring Boot is easier to manage and better-suited for building modern REST applications.

Deployment efforts

When we first started the project, all of the components including the frontend, backend, and MySQL database, were developed and run locally. Once we had a strong starting point for our project, one of our members started to look into Docker and how we could create Dockerfiles specifically for the frontend and backend. This is because we knew we would need these files for our *docker-compose.yml* later on. To create the Dockerfiles for the frontend and backend, we ensured to include all the steps and commands that we use to run the application locally, and we also did some research on the format of standard dockerfiles for Next.js and Spring Boot applications.

Using these Dockerfiles, we then created a *docker-compose.yml* file to connect the frontend, backend, and MySQL database together when running in Docker. To help create this file, we had to do some research on what the file actually entails and the fields we need to include such as environment variables, port mappings, etc. We also created Docker images for both the frontend and backend and then pushed to Docker Hub so they could be hosted publicly. This whole process allowed the entire application to be started using a single command: "*docker-compose up"*.

We chose to deploy our application using Docker because many modern deployment platforms require Dockerfiles as a starting point for deployment. Since the process of creating Dockerfiles for our frontend and backend felt very intuitive, we felt that learning how to configure the *docker-compose.yml* file would be manageable as well.

After successfully deploying the application using Docker, we considered deploying the project to a public URL using cloud deployment services such as Railway specifically for the backend. Our plan was to use the deployed backend URL as the API endpoint when deploying the frontend on a platform such as Vercel. However, we quickly realized that deploying our local MySQL instance would be challenging without us having to migrate to an online SQL database service.

Even though some platforms offer free tiers for hosted SQL databases, we did not have enough time to fully test their reliability. As a result, we decided to keep using Docker for deployment, since it allowed us to run the entire application in a reliable environment. In the future, we hope to migrate our local MySQL database to a publicly hosted database service, which would then allow us to deploy both the frontend and backend to public URLs.

Conclusion

In this project, we applied a layered architecture for the backend using Spring Boot to structure the models, controllers, services, and repositories. This structure separates concerns and improves maintainability. We also used design patterns including MVC and DAO using Spring Data JPA to manage business logic and database access. RESTful APIs and JSON decouple the backend from the presentation layer to make the system more flexible and easier to add data dynamically. These elements allowed our team to build a functional web application while using modern development practices.

The main strengths of our team's design and implementation include a clean, well-structured architecture and effective use of the DAO pattern. This approach separates the

business logic from the database and presentation with the following request flow: Client → Controller → Service → Repository → Database. Our weakness is that we did not consider error and exception handling. During development, we encountered numerous bugs, and while most of them were fixed as they were discovered, some errors were not properly handled through exceptions. This left many areas of the system vulnerable and harder to debug.

Overall, the project was successful. We collaborated effectively as a team by dividing responsibilities and supporting each other when someone needed help navigating the technologies. The use of Spring Boot and Docker helped significantly when building the backend development and deployment. However, we did encounter some merging conflicts and internal deadlines when progress depended on another member completing their tasks. Throughout this project, we learned how to design and implement a real-world application using modern frameworks, and gained a deeper appreciation for the importance of a clean structure and effective collaboration. We also gained a stronger understanding of how the different layers of a system interact. By completing it as a team, there were many advantages, including a shared workload, different design ideas and collaborative problem-solving. However, there were drawbacks, including issues with merging and dependency on others. Despite these challenges, working as a team on this project enhanced the creative process and our overall learning experience.