**SWENG 837 Software Design Final Project**

By Matthew Varghese

Video link: <https://psu.zoom.us/rec/share/hR2gL6MY2njOwPCNUhCj50AnYKMNcWH_WyojfESjKJRTNSJSJ7-pIip97tQrcAaK.sbSNMpZw2KYGmFLI?startTime=1740876052000>

Project Outline

The system I'm building is an e-commerce platform, and it’s designed to provide customers with an easy and secure shopping experience. The system has some functions like user authentication, product browsing, cart management, and a secure checkout process. The goal is it should be simple to create an account, log in, view products, add them to their cart, and check out without any issues. Security is a top priority, especially for handling customer data and payment processing, so I’ll be using Stripe for payments and encrypting sensitive data to keep everything secure. I’ve already built this system in Sweng 861 so I have a lot of the information and background already thought through.

The main goal of the system is to provide a smooth, reliable shopping experience for customers. For performance, I need the platform to handle multiple users so the system has to be scalable if the number of users and products grows. The user interface is simple and responsive and provides real-time updates when items are added to the cart or when product details change. Scalability is very important especially as the platform expands and the Mongodb database needs to be able to handle large amounts of data efficiently. Mongodb is reliable for security as well as response time. Stripe is also an accredited payment site so using that for my secure checkout will enable a level of safety and relief for anyone that makes a purchase.

For security purposes, payment information is encrypted and stored securely. Users should have a secure login experience, and payments should be processed without any issues. In terms of non-functional requirements, it’s also important that the system performs well even with a large number of users. I want to make sure that the platform can scale easily without slowing down as traffic increases, and that downtime is minimized.

System Design using Domain Modeling

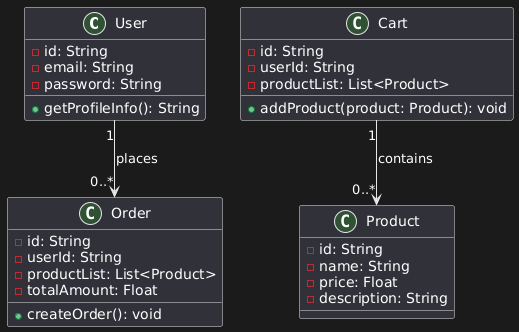
1. UML Use Case Diagram:



**Customer** interacts with the system by doing actions like logging in, browsing products, adding products to the cart, and proceeding to checkout. These are the usual interactions a customer with any kind of ecommerce system.

**Admin**: The admin user has a different set of permissions, including modifying products and managing orders. The Admin would be me alone.

**Stripe** is an external system used for making payments securely during checkout.

1. UML Domain Model:

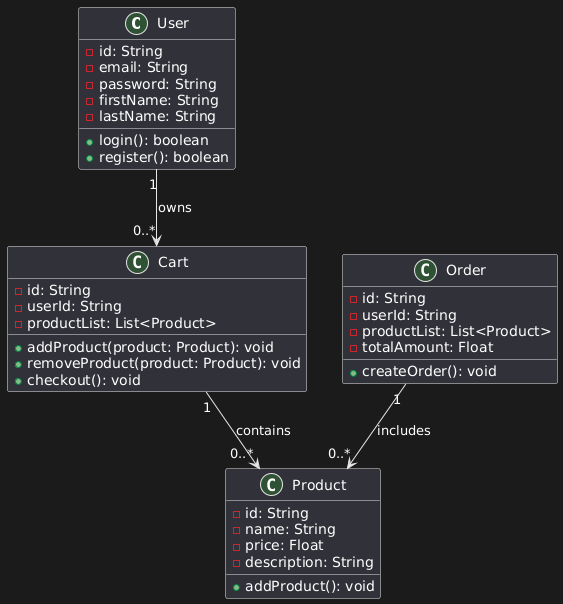
**User** class represents any user of the shop. Each user can have multiple orders and one cart. This is standard for an online shop.

**Product**s represents an item in the clothing store. Products can be added to both the cart and the order.

**Cart** contains the list of products that a user is going to buy. It is associated with a user with their login and it can have multiple products.

**Order** is for when a user completes the checkout process, an order is created. An order contains a list of products and keeps track of the total price.

1. UML Class Diagram:

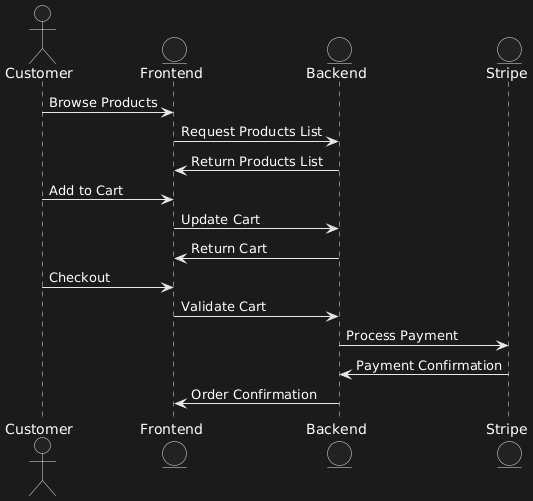


**User Class**: Stores user data such as id, email, password, and profile information. Its methods include login(), register(), and updateProfile().

**Product Class**: Represents individual products with attributes like id, name, price, and description. It's methods are addProduct(), updateProduct(), and deleteProduct() allow product management.

**Cart Class**: Contains a list of products selected by the user. The methods are addProduct(), removeProduct(), and checkout() handle cart operations.

**Order Class**: Once the checkout process is complete, an order is created. It contains the list of products in the order, and the total amount. The methods are createOrder(), cancelOrder(), and viewOrder().

1. UML Sequence Diagrams:  
   

**Customer** starts interacting with the Frontend and then requests product details from the Backend. The customer can also add products to the cart and then go to checkout.

**Frontend** sends updates to the Backend for cart information, order validation, and checkout procedure.

**Backend** validates the cart, calculates total price, and calls Stripe to make the payment. Once payment is confirmed, the Backend finalizes the order and confirms the process with the Frontend.

**Stripe** handles the payment methods and sends the payment confirmation to the Backend.

1. UML State Diagram:

**New**

The order is created but payment is still pending from stripe.

**Payment Pending**

The payment is being processed with stripe.

**Paid**

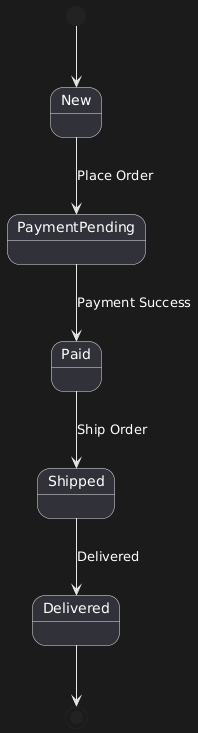
The payment is successful and the order is ready for shipping.

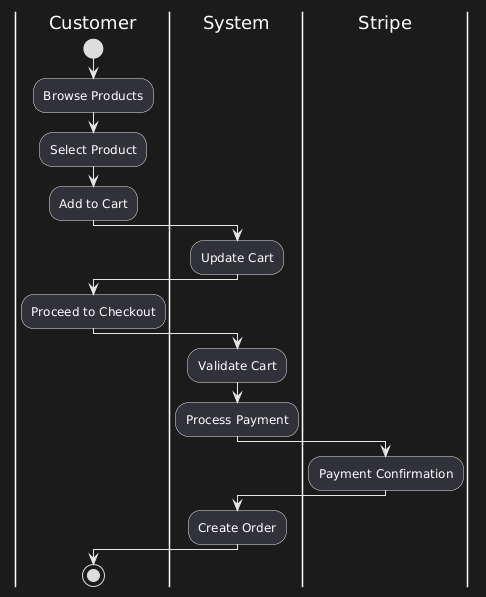
**Shipped**

The order is shipped to the customer.

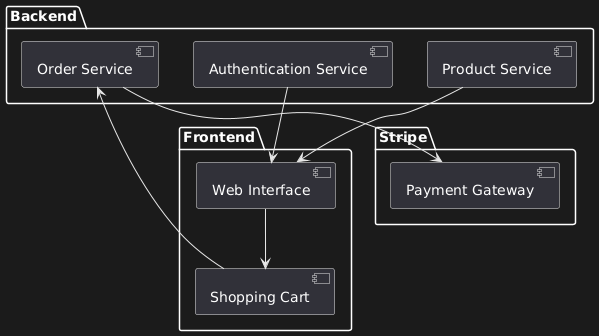
**Delivered**

The order is delivered to the customer.



1. UML Activity Diagram (Swimlane Diagram):  
   

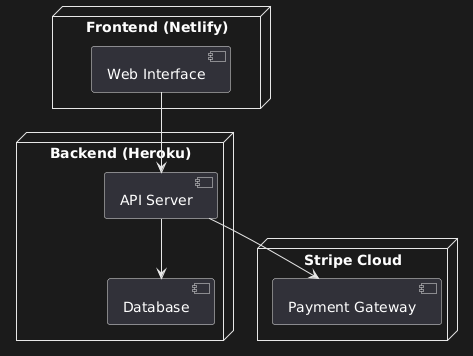
Theres 3 parts for this diagram in swimlanes. **Customer** browses the products, adds them to the cart, and then proceeds to checkout on stripe. **System** will validates the cart, processes payment, and creates the order. **Stripe** will then processes payment and sends back the confirmation to the system

1. UML Component Diagram:

For the component diagram, the **Frontend** handles the user interface, allowing the customer to interact with the website, browse for products, and manage their cart items.

The **Backend** manages the business logic which is user authentication, product management, and order processing.

The **Stripe** component handles payment processing and authentication.

1. Cloud Deployment Diagram:  
   

For the Deployment Diagram, the **Frontend** is hosted on **Netlify** which providing fast and scalable delivery of the user interface. The **Backend** is deployed on **Heroku**, which handles server-side logic and database operations. Heroku **was a great choice for me as it was simple and cheap. It's about 6 dollars a month. The Stripe component** is an external service hosted in the cloud for payment processing.

1. Skeleton Classes and Tables Definition:

* **User Class**: stores the user's information, including attributes like email, password, and other personal information. Methods are login(), register(), and updateProfile()
* **Product Class**: Each product has an id, name, price, and description. There are methods for adding, updating, and removing products, like addProduct(), updateProduct(), and deleteProduct()
* **Cart Class**: This class is just for the cart system. It includes an id, userId, and productList. Methods are addProduct(), removeProduct(), and checkout()
* **Order Class**: Once a customer checks out, an order with Stripe is created. It will have id, userId, productList, and totalAmount. Methods are createOrder(), cancelOrder(), and viewOrder().

1. Design Patterns:

I used various design patterns in the course of this project. This included GRASP, SOLID, GOF, Microservices design patterns that we learned in the course. For GRASP I used the most basic of them which was the expert design pattern. I let the cart class have “responsibility” for the products. The same goes for orders class which handles the orders. For SOLID principles, there were a few that were apparent for each principle. For the single responsibility principle, I could say that the order class only handles order creation and other order matters. It does not have anything to do with other classes. For the open closed principle, I think the project follows this as we could add things by abstraction at any time. An example would be like adding in a new payment system. It would be more of a modification than touching the main source code. For the Integration Segregation principle, theres not really any classes that have unnecessary methods. For the Dependency Inversion principle, there’s instances for example the checkout process depends on the payment service. There were a few microservices I used as well. The order and payment service were both microservices as well as the product service. They are microservices since I used APIs to communicate with them and each other.