UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

"Hermes-Payment App"

Submitted by:

Karan Mangtani	**PES1UG21CS268**
Karthik Namboori	**PES1UG21CS269**
Leharaditya Kumar	**PES1UG21CS300**
M. Nitish	**PES1UG21CS311**

6th Semester E Section

Prof. Course Instructor: Bhargavi Mokashi
Designation: Associate Professor

January - May 2024

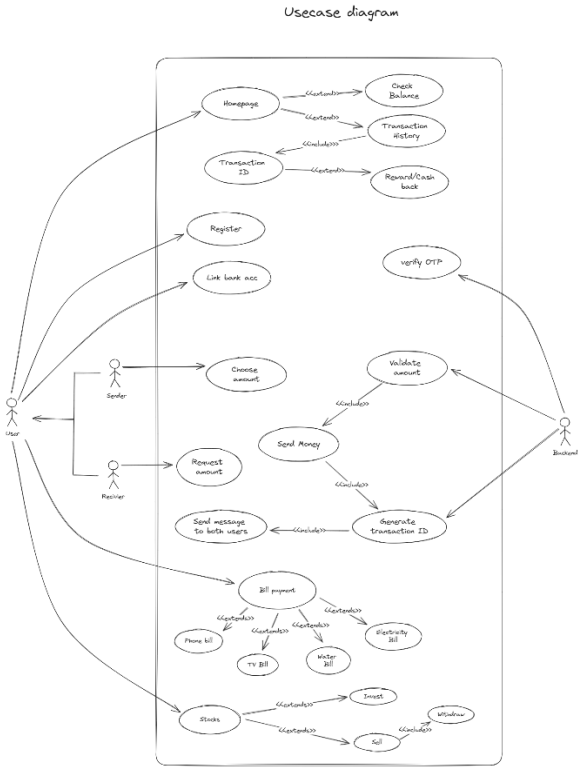
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Problem Statement

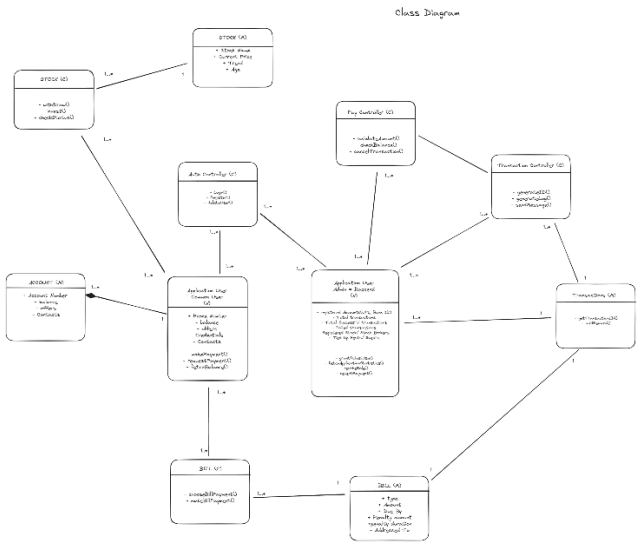
In today's digital landscape, individuals are seeking a consolidated solution for managing their financial transactions and investment portfolios seamlessly. Existing platforms often lack integration between payment processing and stock trading functionalities, resulting in fragmented user experiences and inefficiencies. This creates a pressing need for a comprehensive payments app like Hermes Payments, which offers users the ability to conduct secure transactions, buy and sell stocks, and monitor their investments, all within a single, intuitive interface. By addressing these challenges, Hermes Payments aims to empower users with a unified platform that streamlines financial management and enhances their ability to make informed investment decisions.

Models

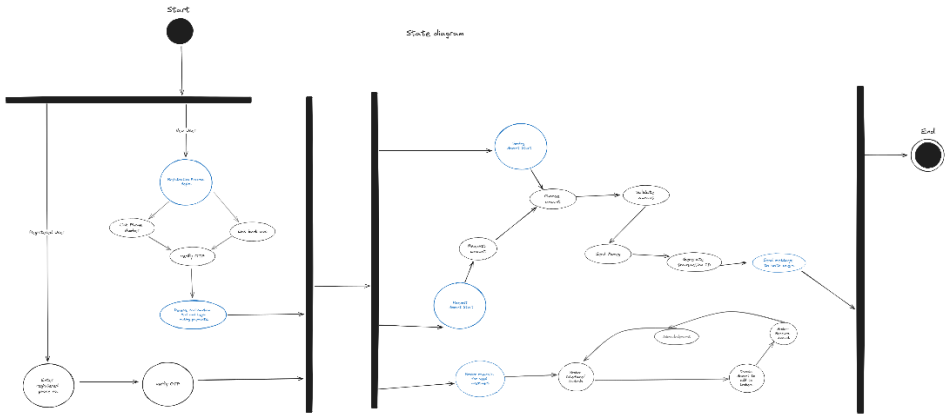
Use Case Diagram:



Class Diagram



State Diagram



Architectural Patterns

The Hermes Payments app adopts the Model-View-Controller (MVC) architectural pattern, ensuring a well-structured and maintainable codebase. In this pattern, the Model layer manages data and business logic, handling tasks such as payment transactions and stock management. The View layer presents the user interface elements to users, facilitating interactions like payments and stock trading. Meanwhile, the Controller layer acts as an intermediary, receiving user input, orchestrating actions within the Model layer, and updating the View accordingly. By separating concerns and promoting modularity, MVC enhances collaboration among developers and ensures a seamless user experience across platforms and devices.

Design Patterns and Principles used

Design Pattern: Observer

- Description: Notify users about changes in their stock portfolio or transaction status in real-time, facilitating seamless updates.
- Java Principle: Polymorphism - Observer establishes a one-to-many dependency, enabling multiple observers to react to subject changes.

Design Pattern: Singleton

- Description: Ensure crucial components like the payment processor are instantiated only once, promoting efficient resource usage and centralized access.
- Java Principle: Encapsulation - Singleton restricts direct instantiation and provides a single point of access, enhancing encapsulation.

Design Pattern: Factory Method

- Description: Create instances of payment methods dynamically based on user preferences, fostering flexibility and extensibility.
- Java Principle: Abstraction - Factory Method defines an interface for object creation, allowing subclasses to implement without exposing instantiation logic.

Design Pattern: Builder

- Description: Use the Builder pattern to construct complex payment transactions or stock trading orders step by step, allowing for the creation of flexible and customizable objects.
- Java Principle: Encapsulation - Builder encapsulates the construction process of complex objects, providing a clear and concise way to create instances with varied configurations without exposing the construction details.

Design Pattern: Adaptor

- Description: Employ the Adapter pattern to convert the interface of existing payment gateways or external stock market APIs into a format that Hermes Payments app can utilize, ensuring seamless integration and interoperability.
- Java Principle: Interface Segregation - Adapter promotes interface segregation by allowing the Hermes Payments app to interact with disparate systems through a unified interface, minimizing dependencies and enabling flexibility in adapting to external services.

SRP (Single Responsibility Principle):

- Description: Each class or module in the Hermes Payments app should have a single responsibility, meaning it should encapsulate only one aspect of the application's functionality. For example, a Payment Processor class should be responsible solely for processing payment transactions, while a Stock Manager class should handle stock-related operations.

- Benefit: By adhering to SRP, the codebase becomes more modular, easier to understand, and less prone to bugs. Changes in one aspect of the system are less likely to affect unrelated parts, promoting code maintainability and scalability.

OCP (Open/Closed Principle):

- Description: The design of the Hermes Payments app should be open for extension but closed for modification. This means that new features or functionalities can be added to the system without altering existing code. For instance, new payment methods or stock trading strategies can be introduced without modifying the core payment processing or stock management logic.
- Benefit: OCP promotes code extensibility and minimizes the risk of introducing bugs when extending the system. It encourages the use of design patterns like Strategy or Factory Method to achieve flexible and modular designs.

MVC (Model-View-Controller):

- Description: The Hermes Payments app follows the MVC architectural pattern, where the Model represents data and business logic, the View handles user interface elements, and the Controller manages user input and orchestrates interactions between the Model and View layers.
- Benefit: MVC promotes separation of concerns, making the codebase more organized and easier to maintain. It enhances code reusability and modularity, allowing for independent development and testing of each component. Additionally, MVC facilitates collaboration among developers by providing clear boundaries between different parts of the application.

Low Coupling (GRASP):

- Description: Low coupling in the Hermes Payments app refers to reducing the dependencies between classes or modules. Each component should rely on abstractions or interfaces rather than concrete implementations, allowing for flexibility and easier modification.
- Benefit: Low coupling improves the system's resilience to changes, as modifications to one component are less likely to impact others. It promotes code reuse, as components can be easily swapped or replaced without affecting the overall system functionality. Additionally, low coupling enhances testability by facilitating the isolation of components for unit testing.

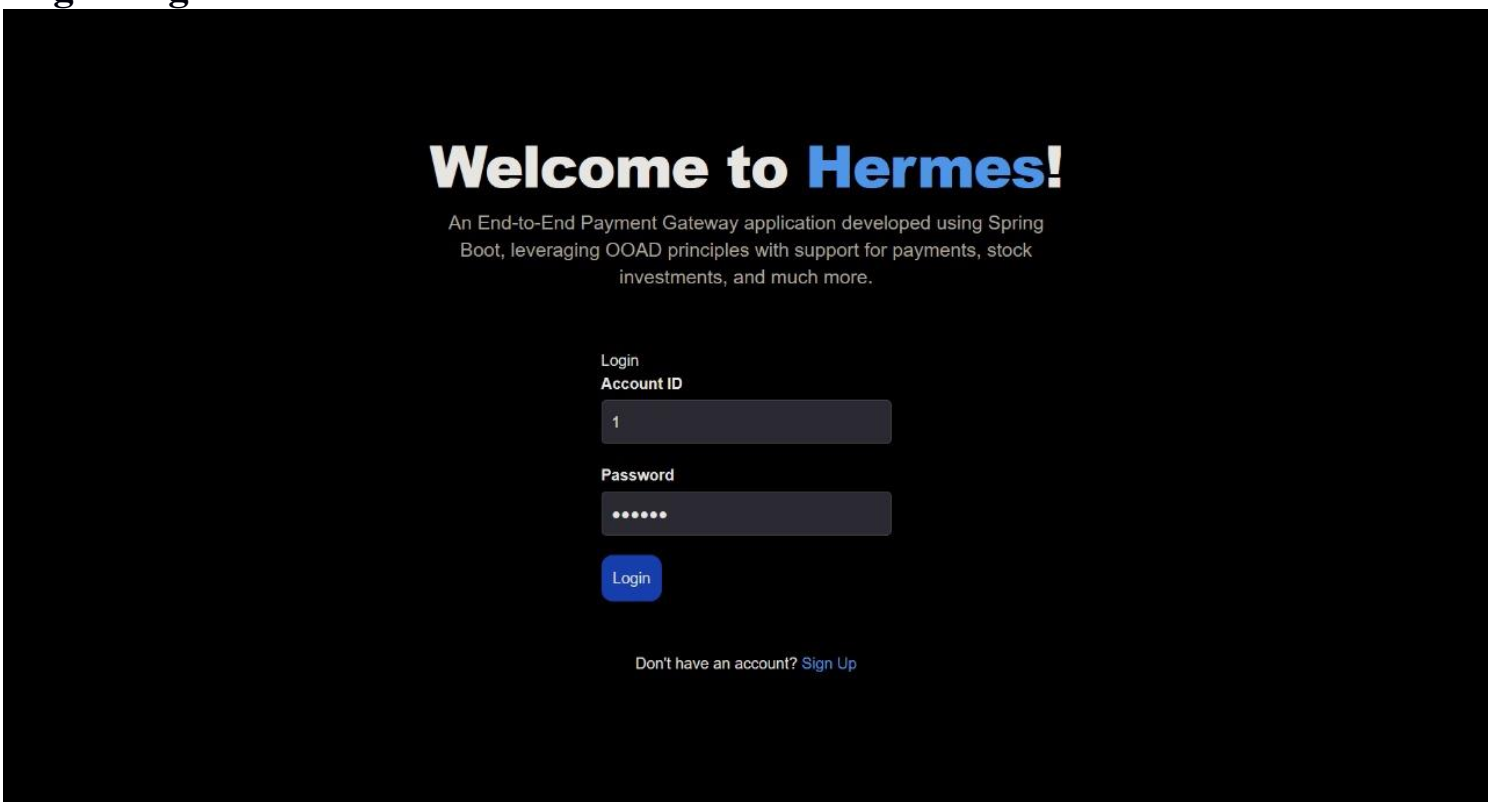
Creator (GRASP):

- Description: The Creator principle in the Hermes Payments app dictates that classes should be responsible for creating instances of other classes with which they have a strong association or ownership. For example, a Payment Processor class may create instances of Payment objects, and a Stock Manager class may create instances of Stock objects.

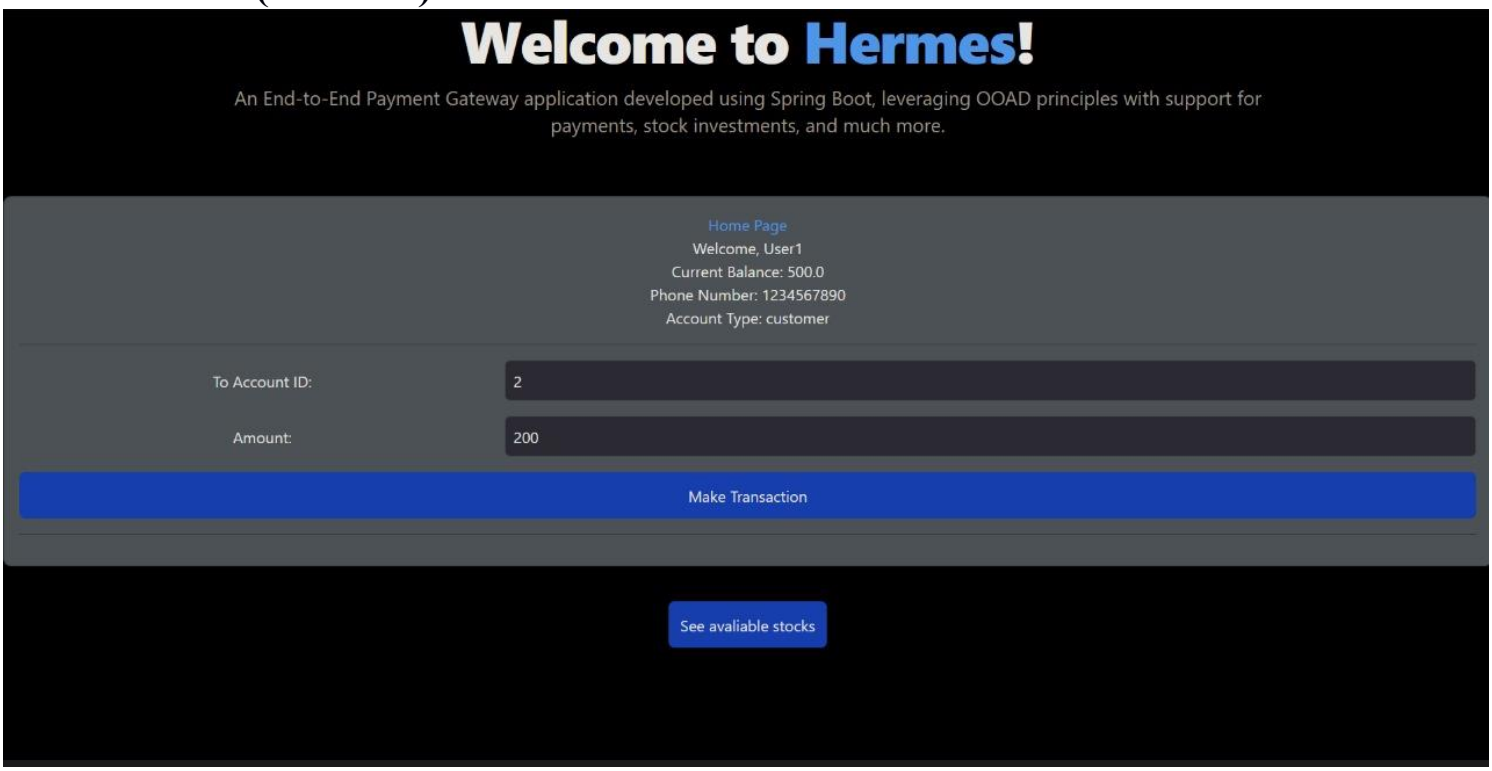
- Benefit: By adhering to the Creator principle, the codebase becomes more cohesive and easier to maintain. It ensures that object creation logic is centralized within the appropriate classes, reducing redundancy and promoting consistency. Additionally, it simplifies the process of tracking and managing object creation throughout the application.

Screenshots

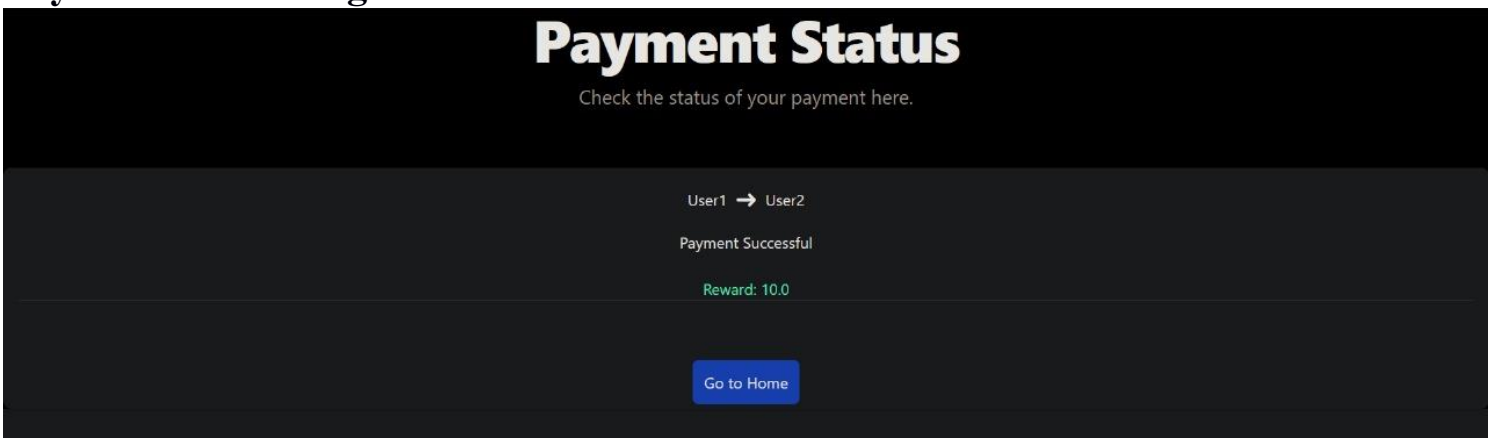
Login Page:



Home Screen (for User):

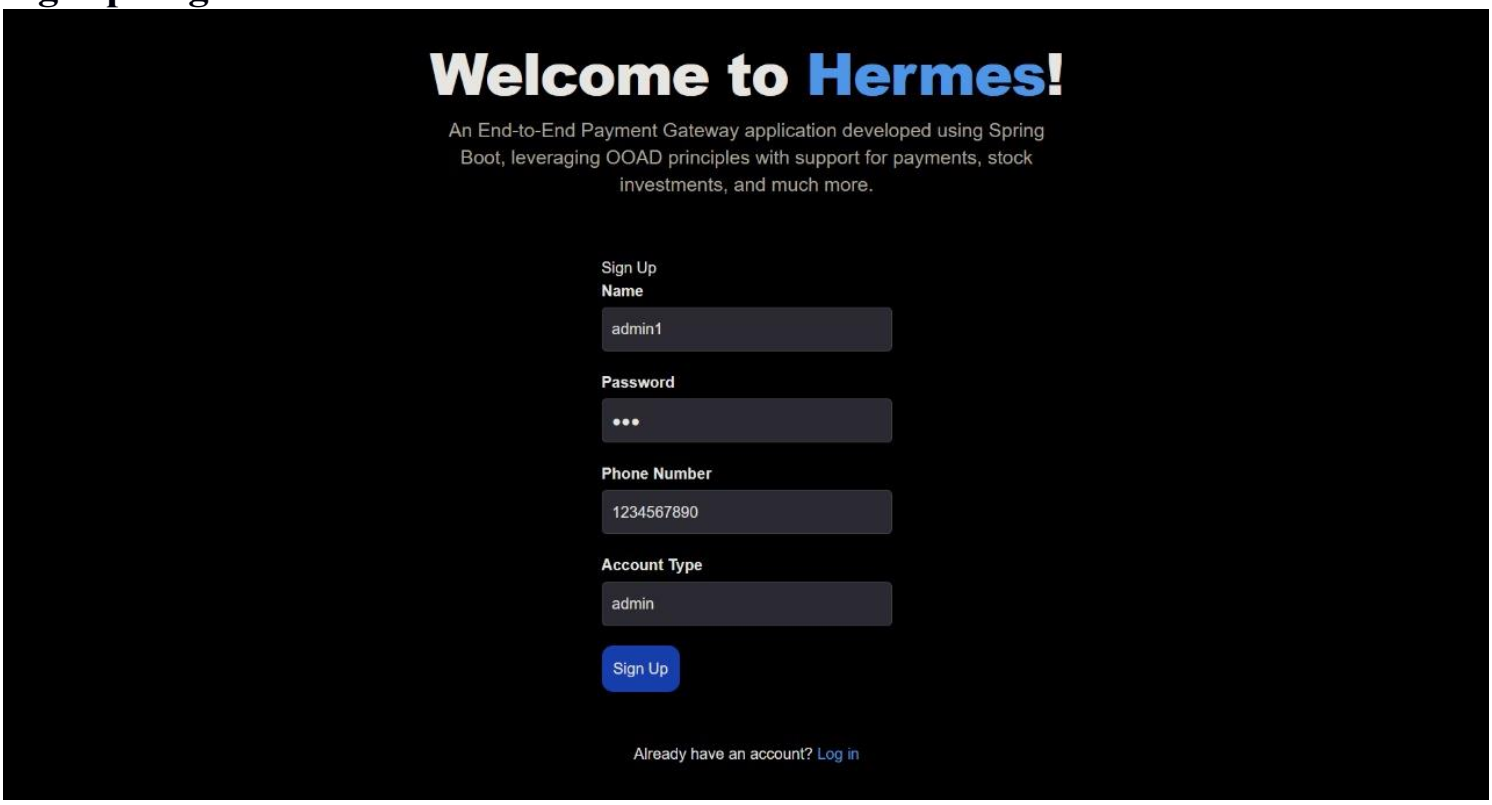


Payment Status Page:

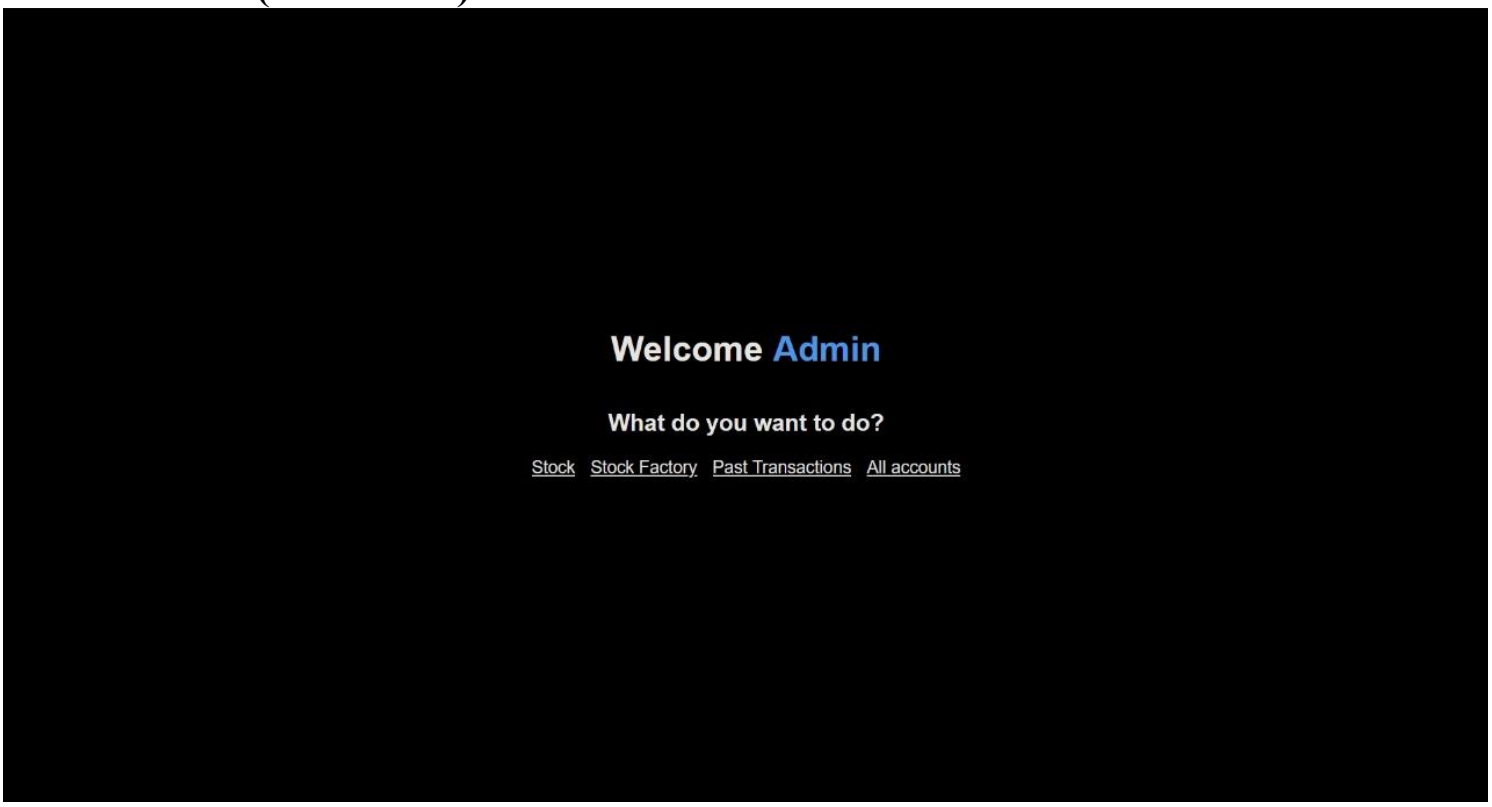


The reward system is set up for the payment done to merchant

Signup Page:



Home Screen (for Admin):



Accounts list (for Admins):

Account List

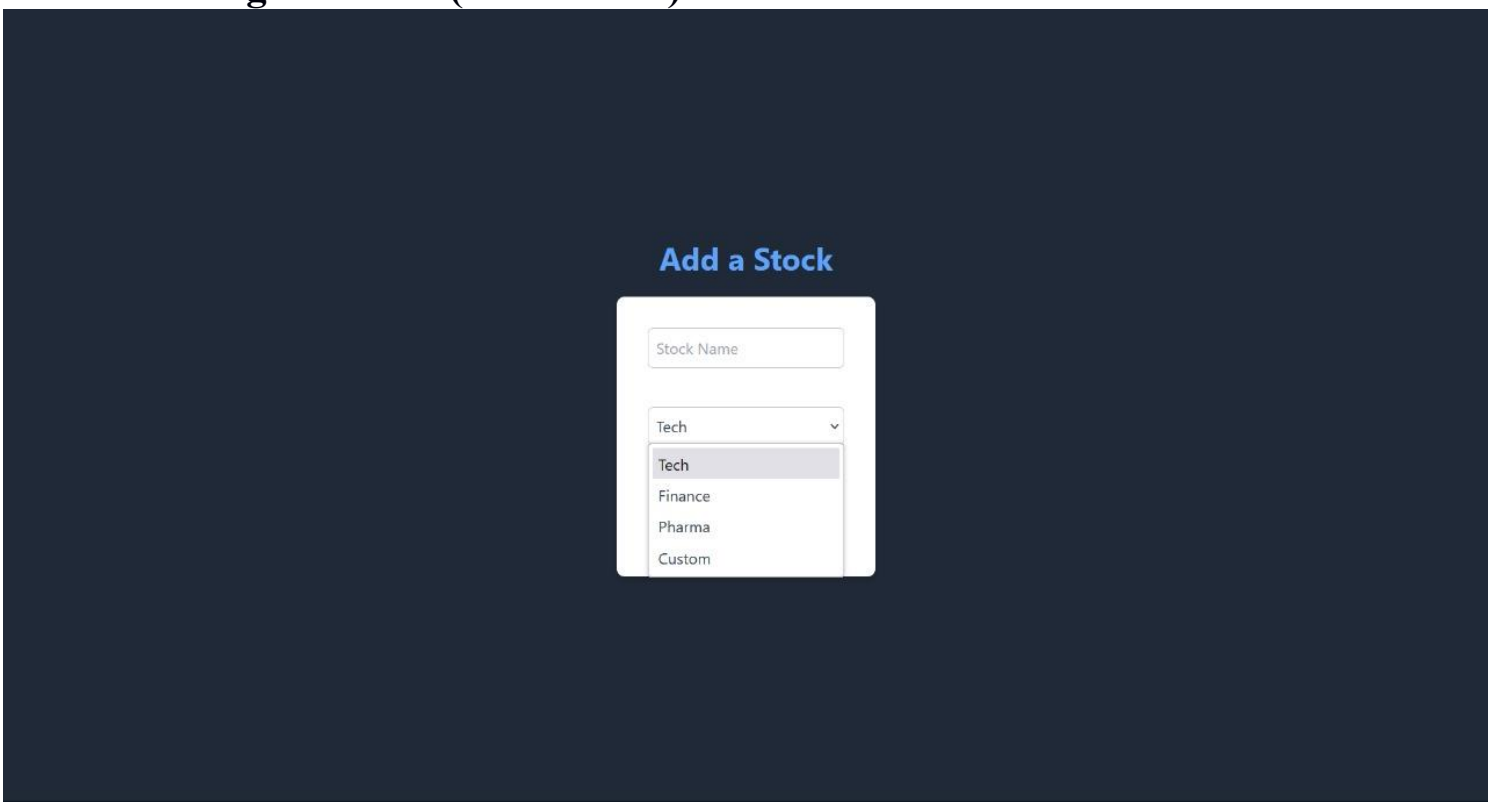
ID	Name	Phone Number	Balance	Account Type
2	User2	1234567899	1000.0	merchant
1	User1	1234567890	310.0	customer
3	admin1	1234567890	500.0	admin
4	admin2	1234567899	500.0	admin

Transactions list (for Admins):

Transaction List

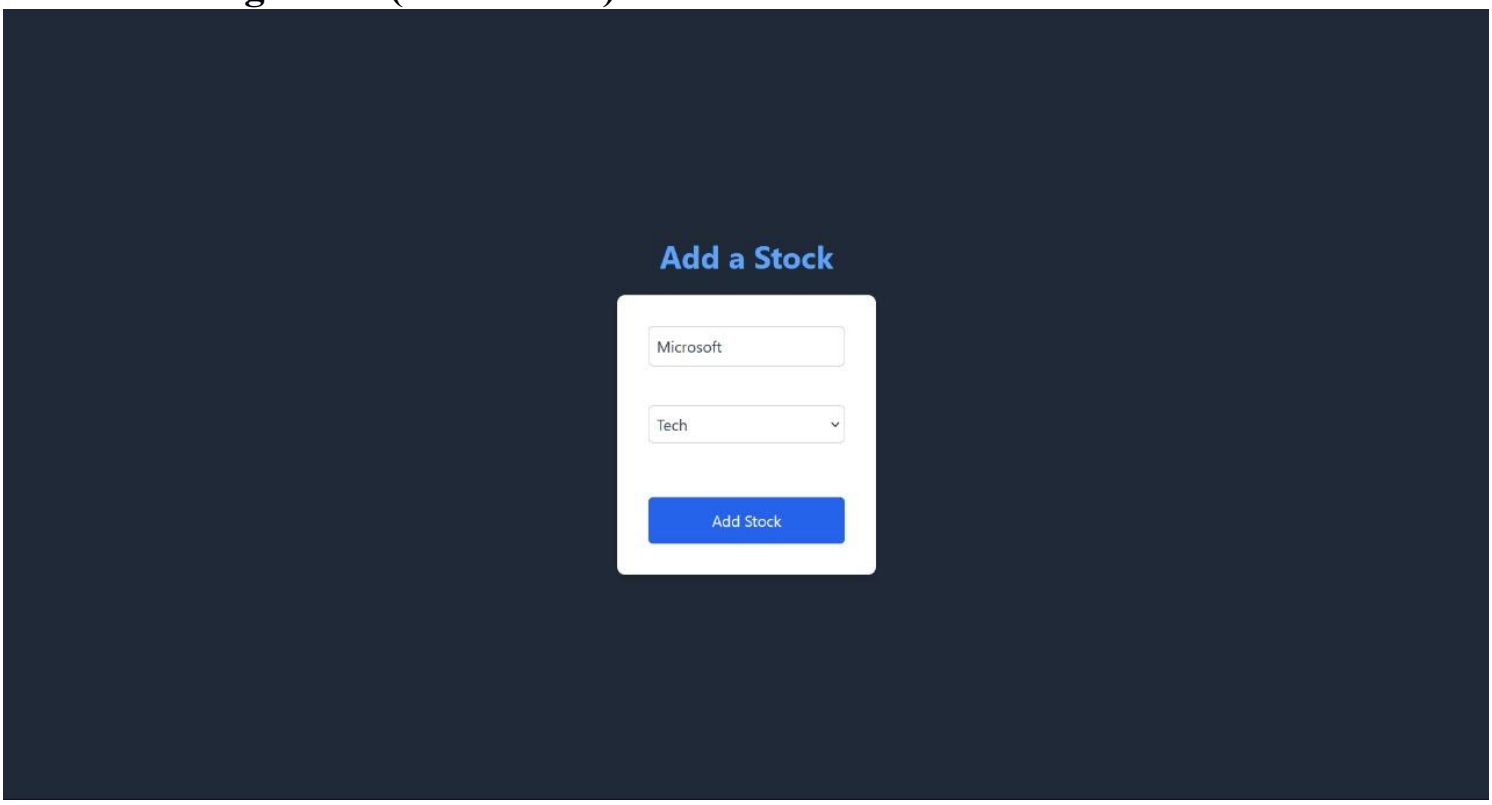
ID	Amount	Time	From Account ID	To Account ID
1	200.0	2024-04-30 20:05:02.269995	1	2

Stock building/ creation (for Admins):

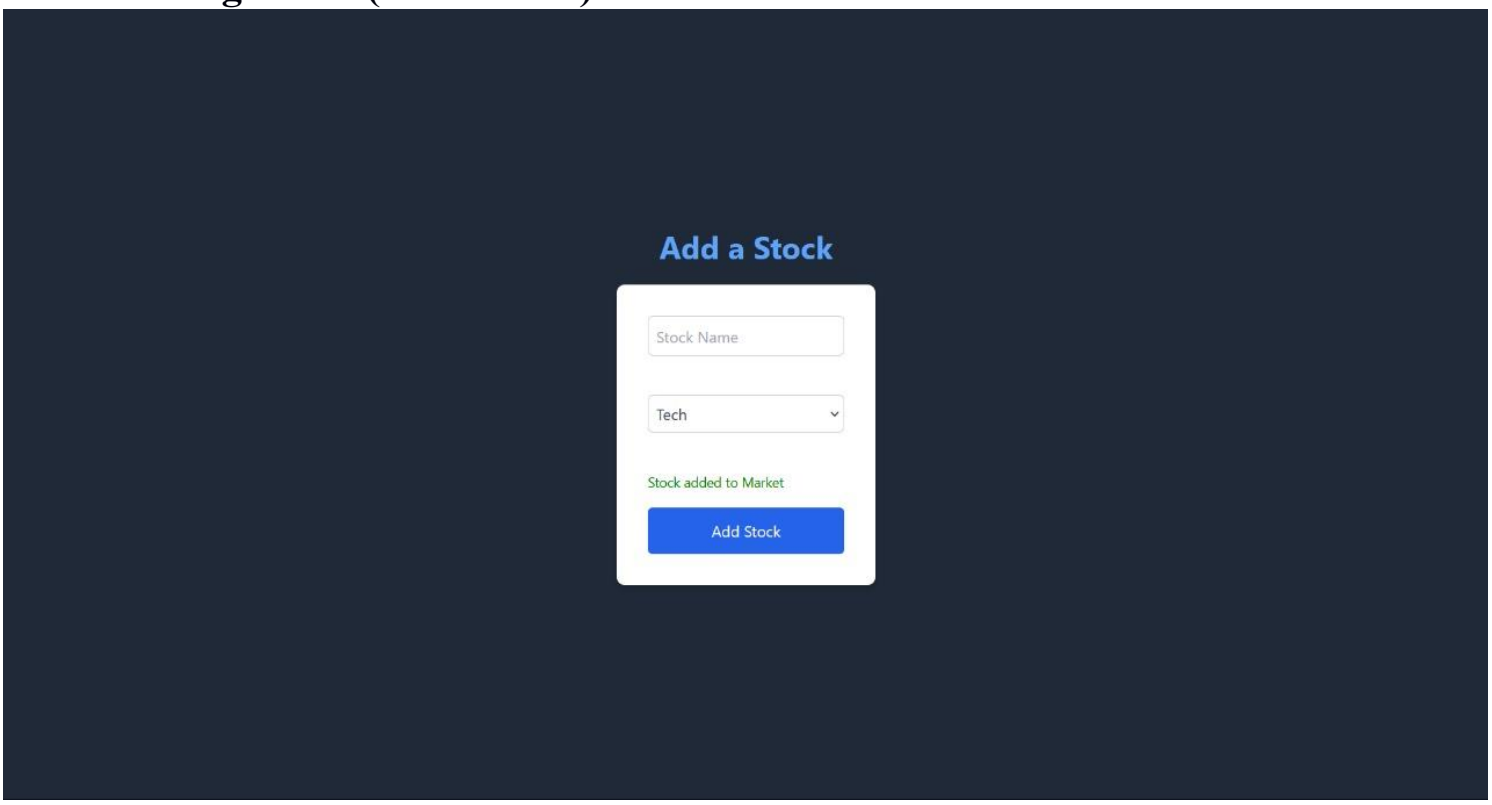


In above diagram, Tech, Finance and Pharma are using Factory Pattern and the custom is using builder Pattern.

Before Adding Stock (For Admin):



After Adding Stock (For Admin):



All Stock List (For Admin):

Stock List

Name	Type	Price
Microsoft	tech	150.0
Yokohama	custom	107.0

Subscribers are notified (Subscribers here are the all the admin accounts and they subscribe to the topic 'transactions' and whenever any transaction takes place then the admins are notified as seen in the terminal):

```
----------------
Notifying subs ...
----------------
Admin with ID 3 and name admin1 was informed about the transaction.
----------------
----------------
Admin with ID 4 and name admin2 was informed about the transaction.
----------------
```

Postgres database:

```
upi=# select * from account;
 balance | account_id | id | account_type |  name  | password |  phoneno
---------+------------+----+--------------+--------+----------+------------
     500 |          3 |  3 | admin        | admin1 | qwe      | 1234567890
     500 |          4 |  4 | admin        | admin2 | qwe      | 1234567899
     500 |          2 |  2 | merchant     | User2  | 12345    | 1234567899
     810 |          1 |  1 | customer     | User1  | qwerty   | 1234567890
(4 rows)


upi=# select * from stock;
 price | id |   name    |  type
-------+----+-----------+--------
   150 |  1 | Microsoft | tech
   107 |  2 | Yokohama  | custom
(2 rows)


upi=# select * from transaction;
 amount | from_account_id | id |            time            | to_account_id
--------+-----------------+----+----------------------------+---------------
    200 |               1 |  1 | 2024-04-30 20:05:02.269995 |             2
    500 |               2 |  2 | 2024-04-30 20:09:23.307089 |             1
(2 rows)
```