UE22CS352B - Object Oriented Analysis & Design

Mini Project Report

Title
Design and Implementation of a Pharmacy Management System

Submitted by:

Shereen Anand: PES1UG22CS557
Shraddha: PES1UG22CS565
Shreya Naveen: PES1UG22CS576
Shreya Srikant: PES1UG22CS577

6th semester ***J section***
Semester Section

Faculty Name: Dr. Bhargavi Mokashi

January - May 2025

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 the pharmaceutical sector, effective management of stock, suppliers, customers, and billing.
Operations is critical to ensure uninterrupted service and operational efficiency. Manual processes often result in stock shortages, supplier miscommunications, customer dissatisfaction, and billing inaccuracies, thereby affecting business continuity and customer trust.
The absence of an integrated system leads to challenges such as delayed stock replenishment,
inefficient supplier handling, poor customer relationship management, and revenue loss due to billing errors.
Therefore, there is a need for a comprehensive **Pharmacy Management System** that automates inventory tracking, supplier coordination, customer management, and billing operations to enhance accuracy, efficiency, and service quality.

Key Features:

1. Stock Management

The system keeps track of all medicines and products available in the pharmacy.
It automatically updates stock quantities when sales or new purchases are made.
Low-stock alerts help ensure that important medicines are always available, avoiding shortages.

2. Supplier Management

The system maintains a record of all suppliers along with their contact details and order history.
It allows users to create purchase orders and monitor pending deliveries, making it easier to manage restocking and supplier relationships.

3. Customer Management

Customers' basic information and their purchase histories are stored securely in the system.
This helps in providing better service, tracking customer preferences, and implementing

loyalty programmes if needed.

4. Billing

The system provides a fast and accurate billing process.
It automatically calculates the total amount, applies any discounts, and generates invoices.
It also supports multiple payment options like cash, card, and online transfers, ensuring a smooth checkout experience.

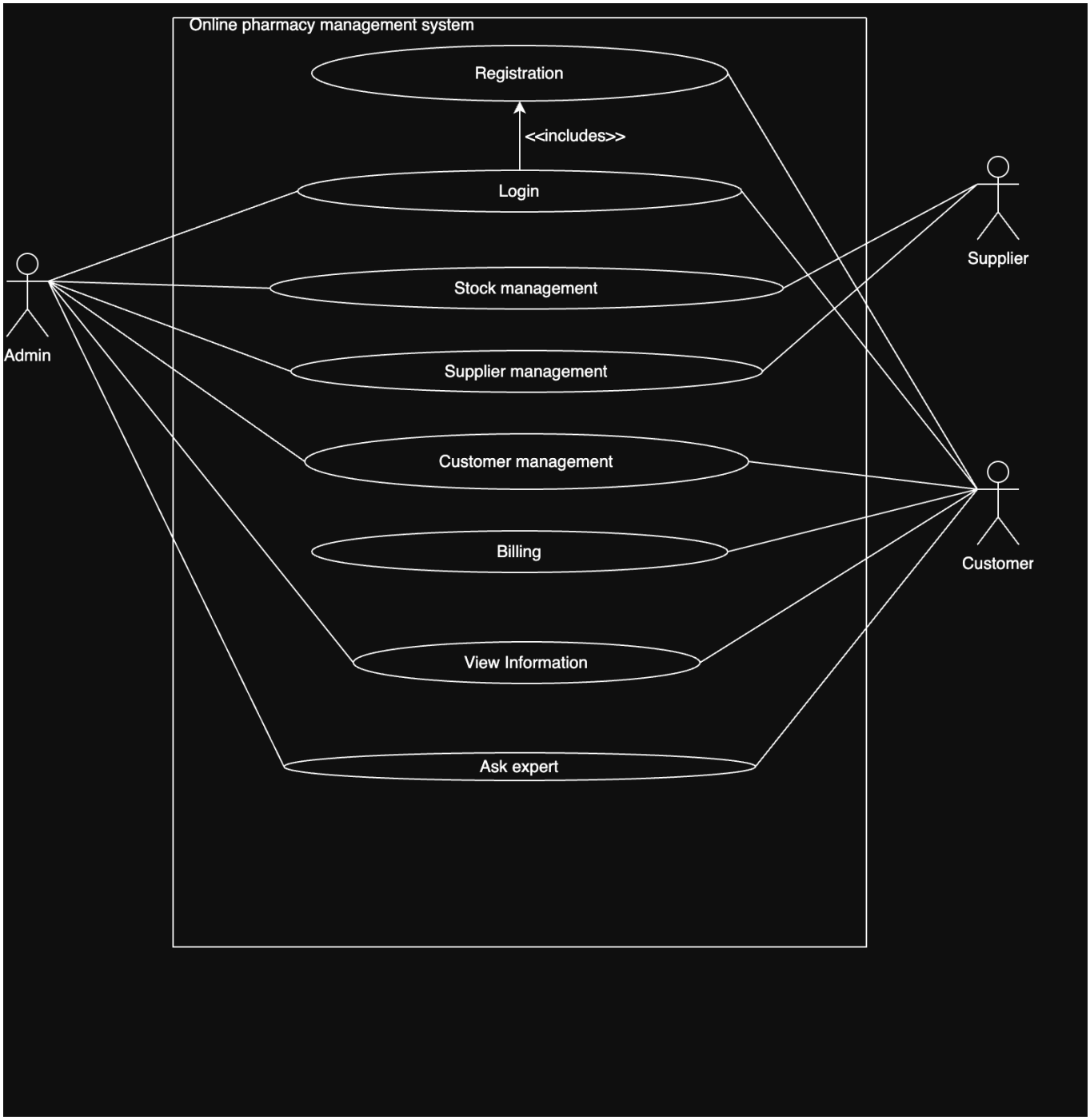
5. User Roles

Different types of users, such as inventory managers, sales staff, and administrators, can log into the system with appropriate access rights. This ensures that only authorised users can perform specific operations like updating stock or generating bills.

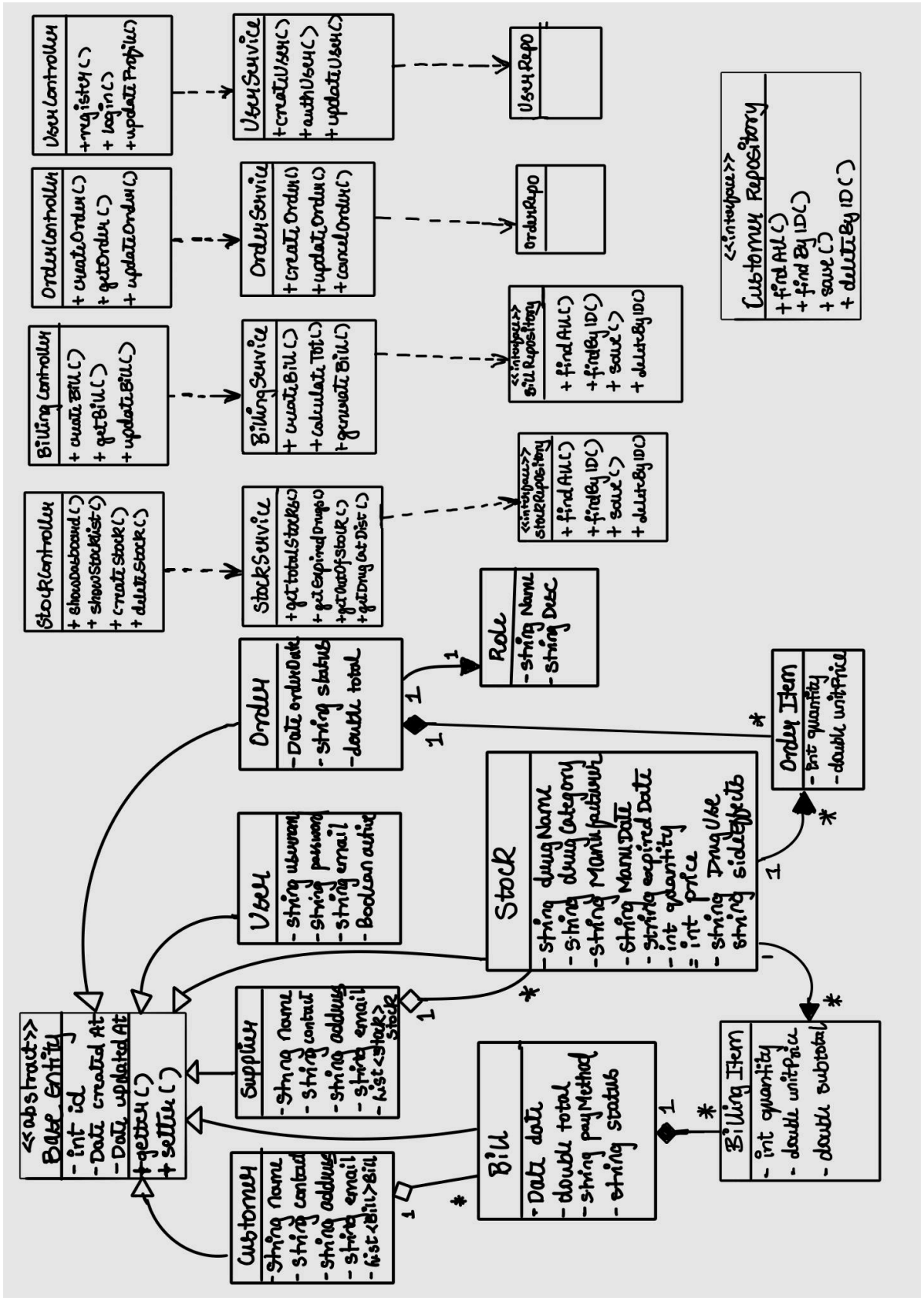
6. Stock report analysis

The dashboard provides a visual analysis of stock distribution by category and manufacturer. This helps in quick inventory assessment and decision making.

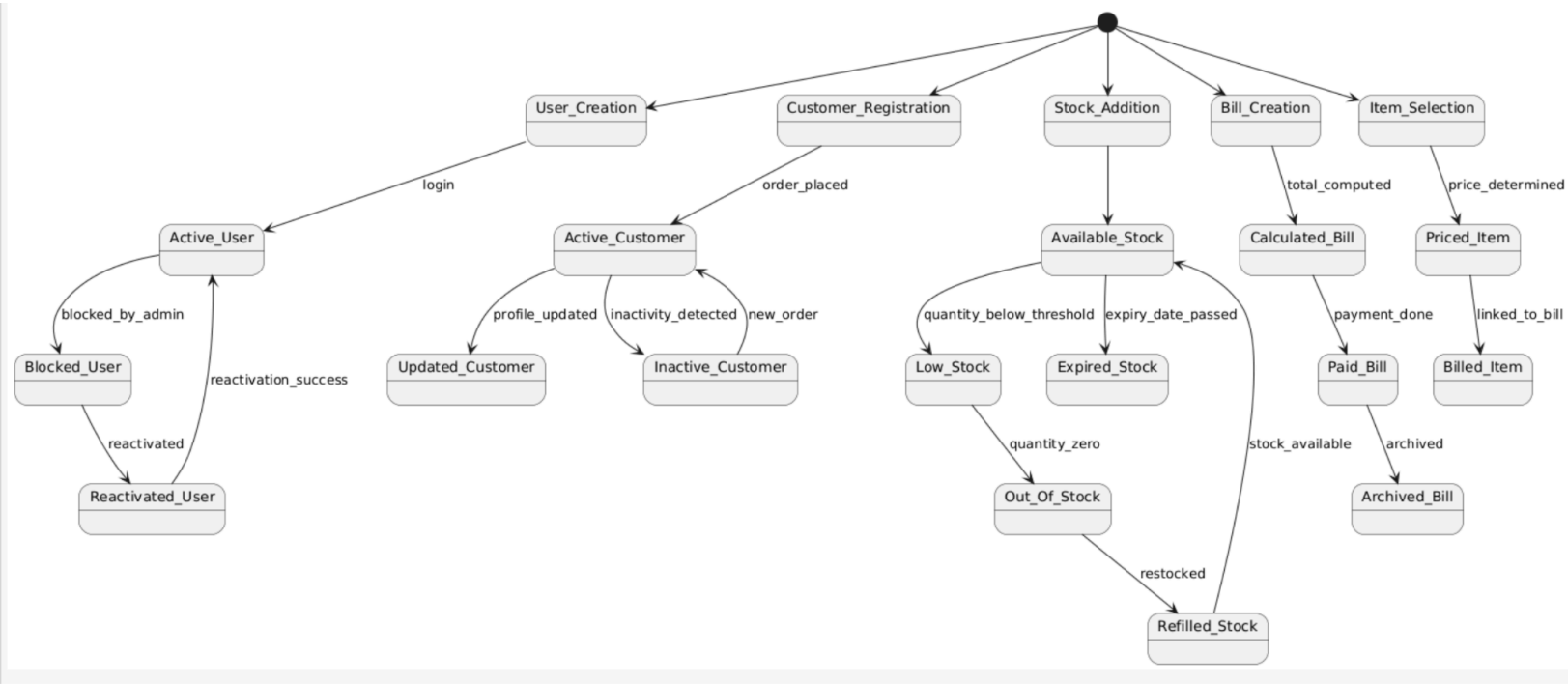
Use Case Diagram:



Class Diagram:

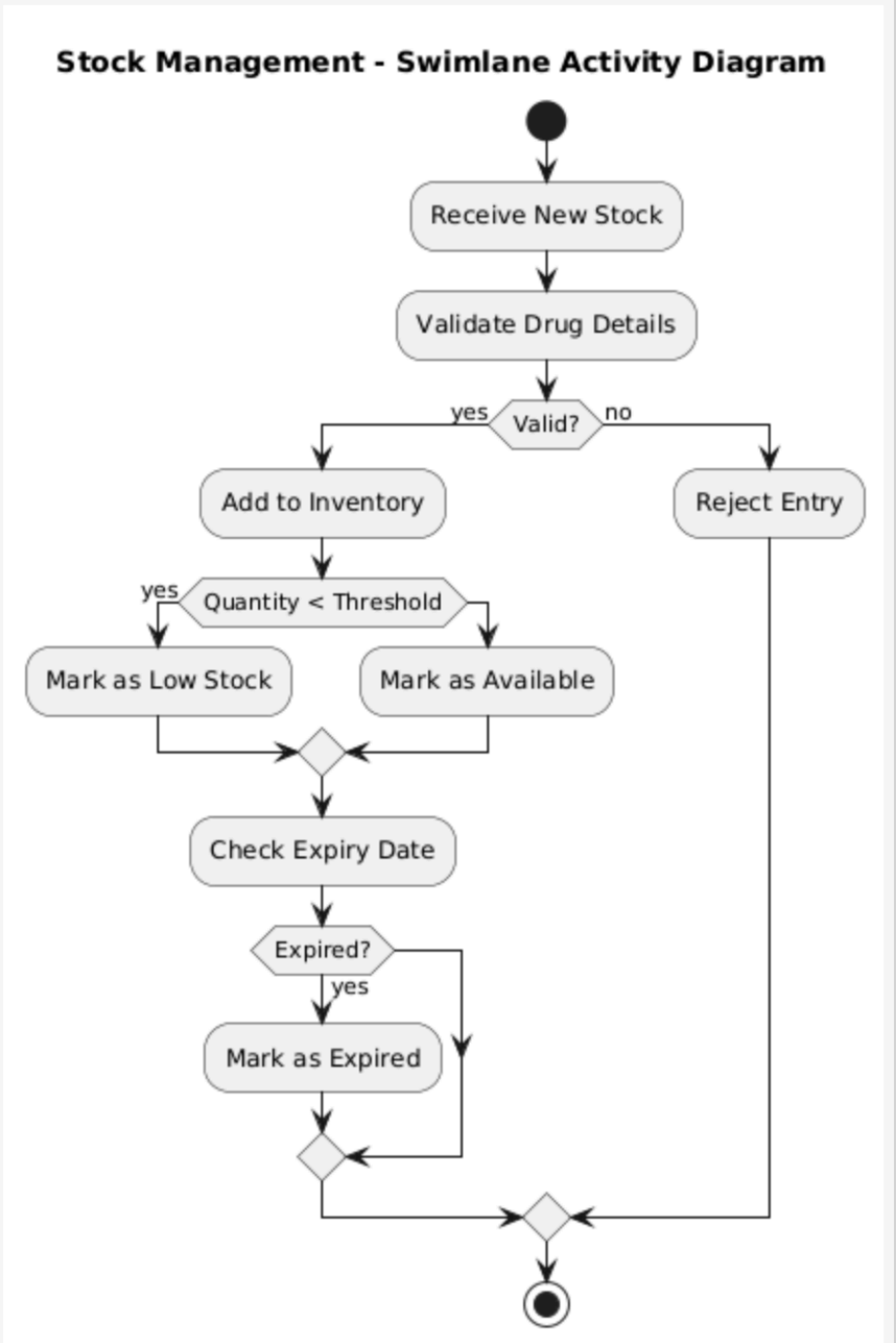


State Diagram:

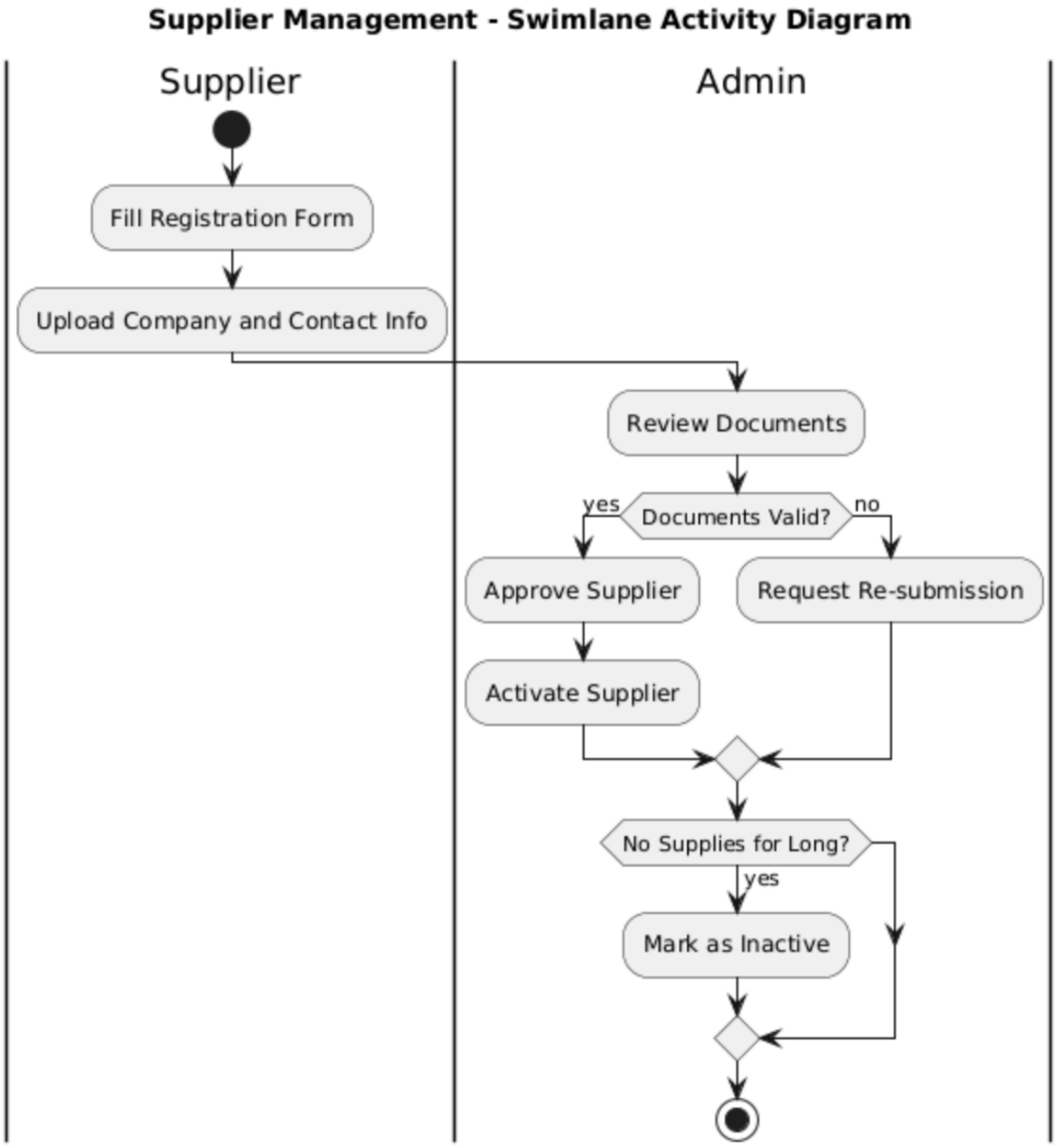


Activity Diagrams: Major Use Cases-

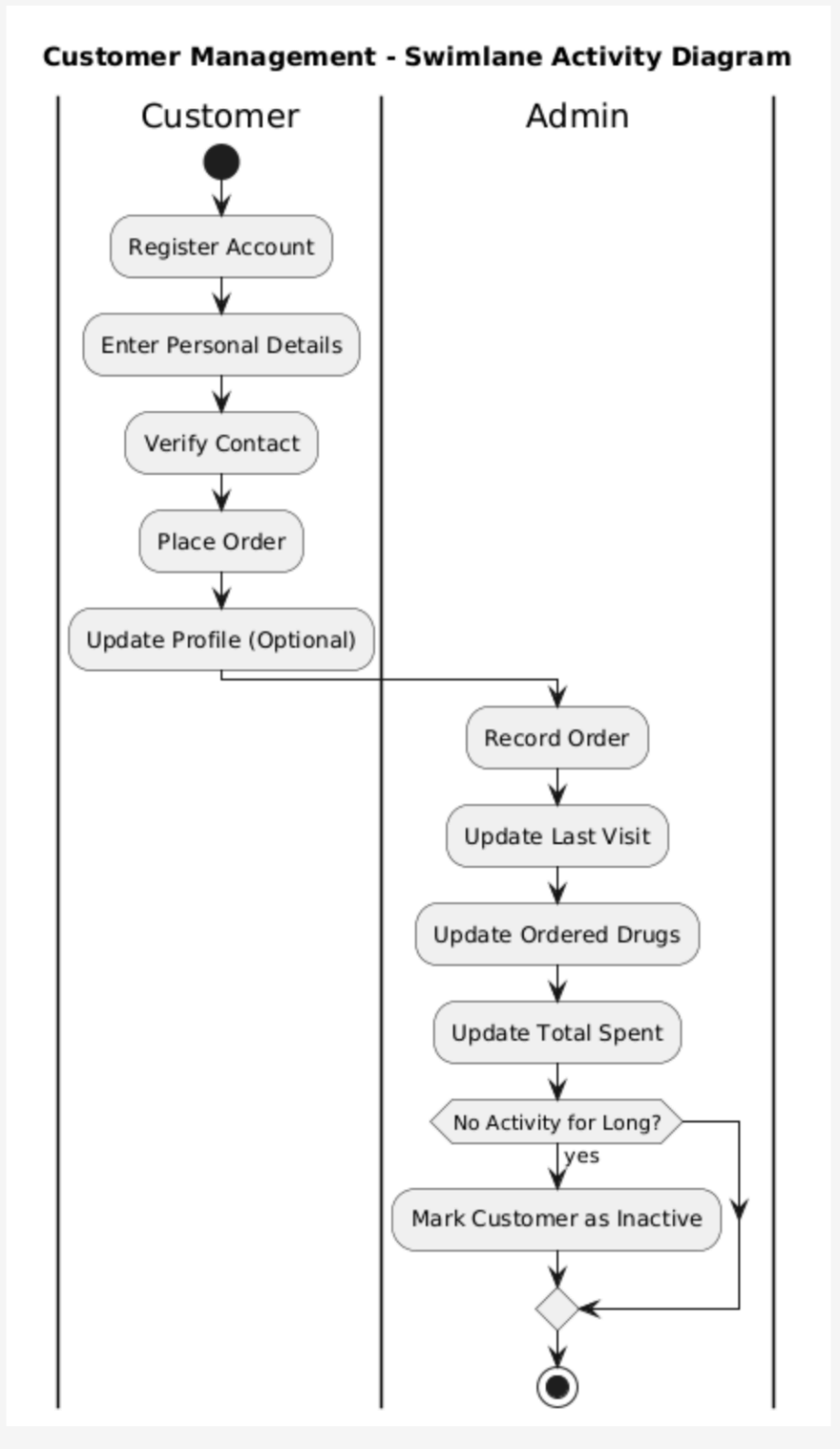
STOCK MANAGEMENT



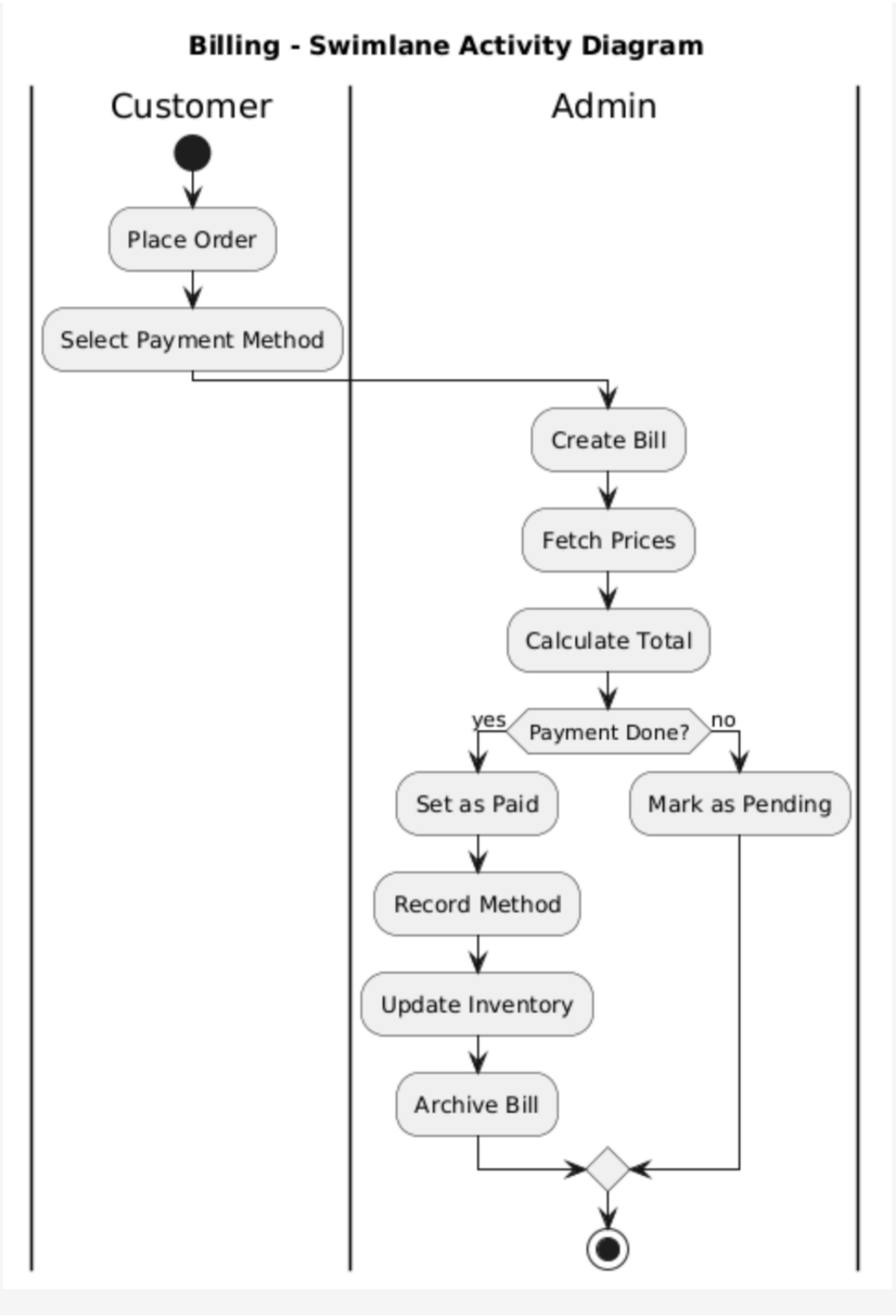
SUPPLIER MANAGEMENT



CUSTOMER MANAGEMENT

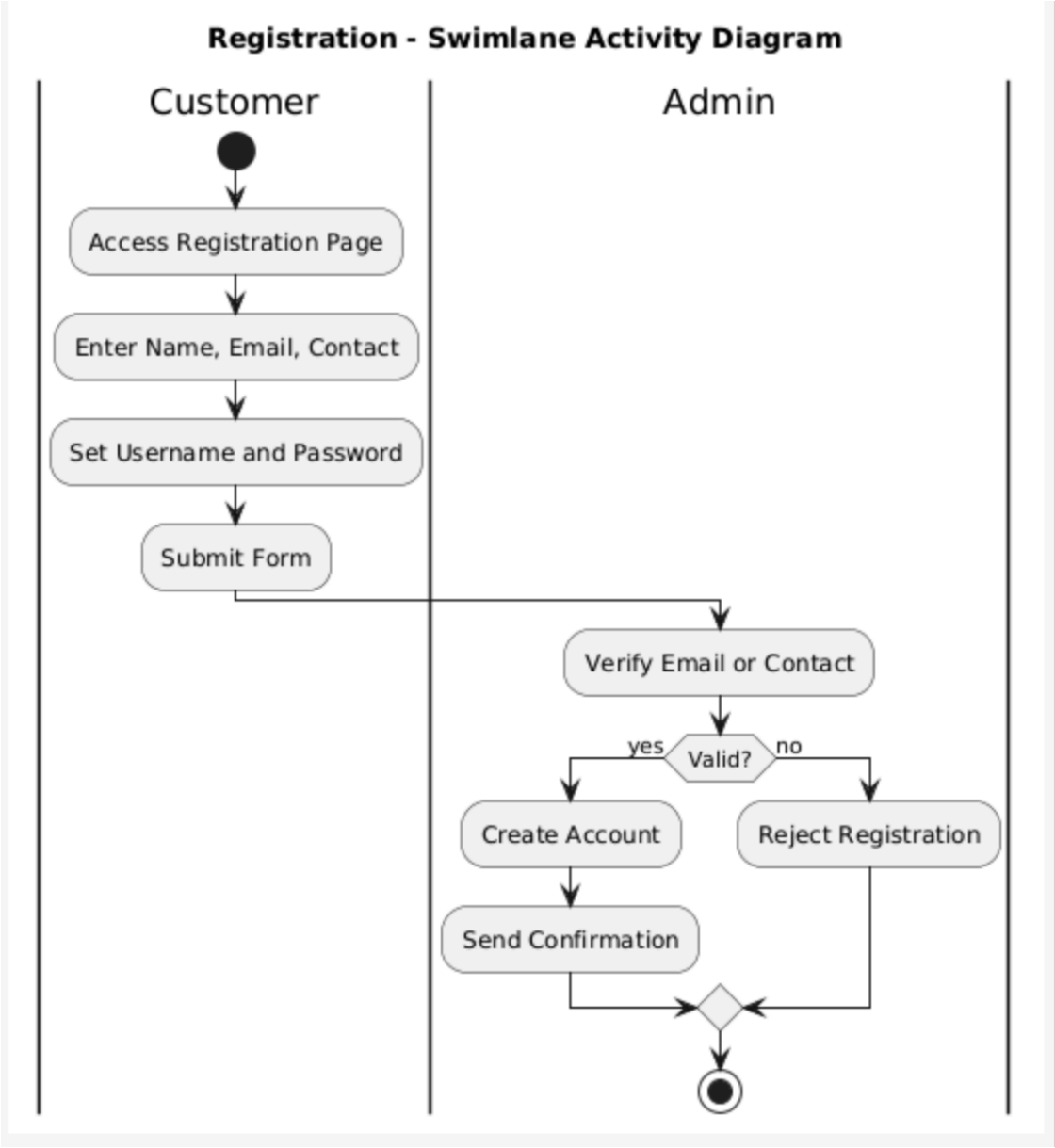


BILLING

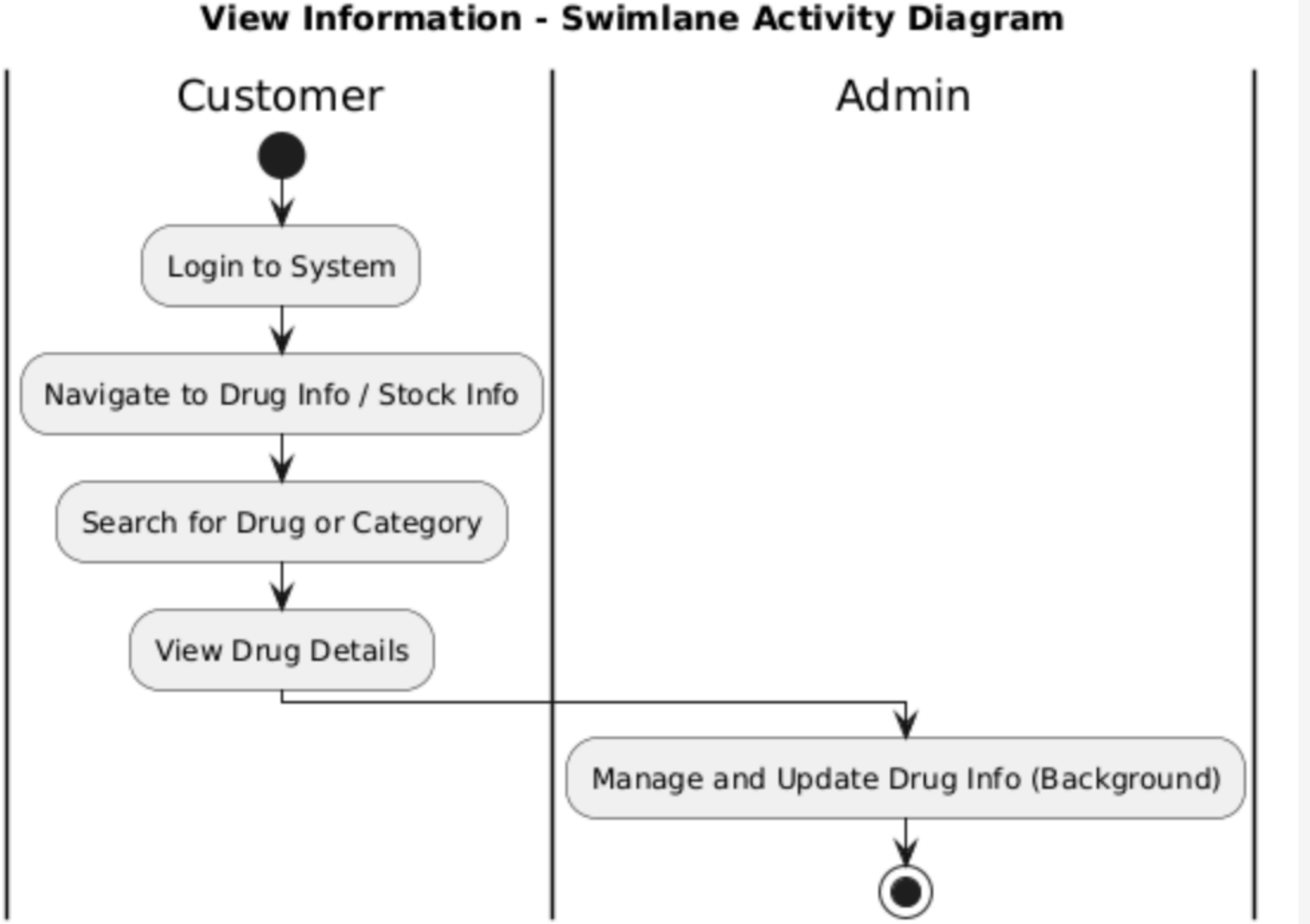

Billing - Swimlane Activity Diagram
Customer
Admin
Place Order
Select Payment Method
Create Bill
Fetch Prices
Calculate Total
yes
Payment Done?
no
Set as Paid
Mark as Pending
Record Method
Update Inventory
Archive Bill

Activity Diagrams: Minor Use Cases-

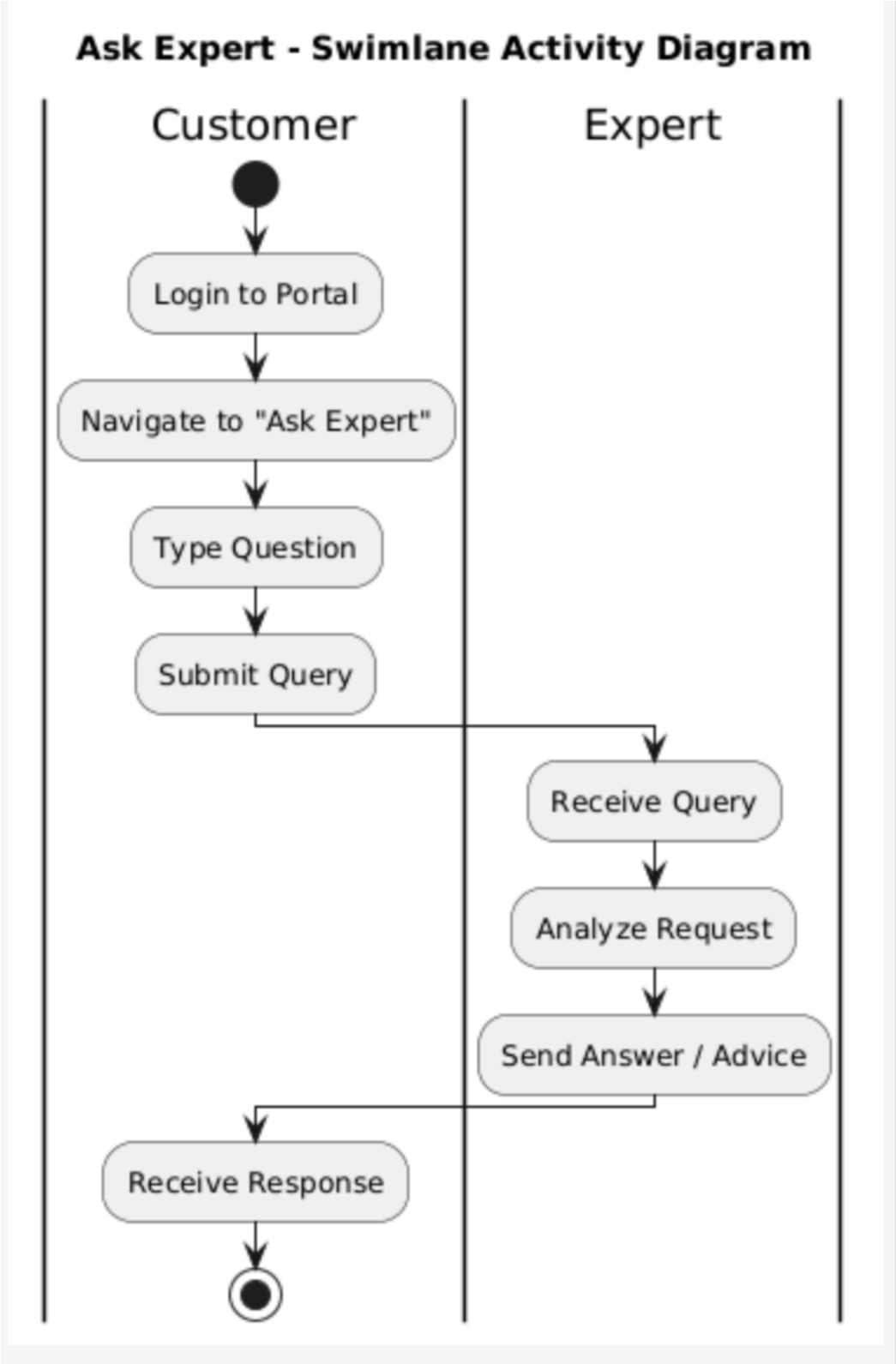
Registration:



View Information:



Ask Expert


Ask Expert - Swimlane Activity Diagram
Customer
Expert
Login to Portal
Navigate to "Ask Expert"
Type Question
Submit Query
Receive Query
Analyze Request
Send Answer / Advice
Receive Response

Architecture Patterns, Design Principles, and Design Patterns:

Architecture Patterns

Model–View–Controller (MVC) Pattern

1. Model Layer

The **model** component is responsible for the core functionality of the application, including data representation and interaction with the database. In this project, the model consists of **Java entity classes** and the corresponding **repository interfaces** for data access.

Example:

- `Stock.java` represents the structure of the stock entity with fields such as `id`, `name`, `quantity`, `price`, and `expiryDate`.
- `StockRepository.java` is an interface, providing CRUD operations without the need for boilerplate SQL code.

The model layer ensures that all data-related logic is encapsulated in a consistent and reusable way.

2. View Layer

The **View** layer is responsible for rendering the user interface and displaying data to the users. In this system, the views are implemented using **Thymeleaf**, a modern server-side Java template engine that allows dynamic rendering of HTML pages based on the

data passed from the controller.

Each HTML file under the `templates/` directory corresponds to a specific functionality. For example, `stock_list.html` displays all stock items in a tabular format, dynamically populated using Thymeleaf expressions such as `${stock.name}`.

The view layer remains entirely separate from the business logic, ensuring that designers and developers can work independently on the UI and backend logic.

3. Controller Layer

The **Controller** serves as the intermediary between the Model and View. It handles incoming HTTP requests from users, processes them using the business logic defined in the service layer or model, and returns the appropriate HTML view.

In this system, controllers are annotated with `@Controller` and mapped to specific URL paths. For instance:

- `StockController.java` handles requests related to stock management, such as viewing, adding, and deleting stock items.
- It uses `@GetMapping` and `@PostMapping` annotations to define endpoints, and uses `Model` to pass data to the view

Design Patterns

1. Facade Pattern – CustomerFacade

What it does: Simplifies interactions with complex subsystems like services and models.

Where it's used: CustomerFacade uses 3 different classes : bill.java, customer.java, order.java and wraps it into a simpler customer management dashboard including information about customer details, order details and billing.

```java
public class CustomerFacade {

    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private BillingService billingService;

    @Autowired
    private OrderService orderService;

    public List<Customer> getAllCustomers() {
        return customerRepository.findAll();
    }

    public Customer getCustomerById(Long id) {
        return customerRepository.findById(id).orElse(other:null);
    }

    public Customer getCustomerByContact(String contact) {
        return customerRepository.findByContact(contact);
    }
```

2. Factory Pattern – UserFactory, UserFactoryImplementation, User

What it does: Handles object creation, making it easy to plug in different billing strategies.

Where it's used: Determines which Role the user will take : admin or customer

```java
package com.pharmacy.Management.factory;

import com.pharmacy.Management.models.User;

public interface UserFactory {
    User createUser(String username, String password, String email);
}
```

```java
package com.pharmacy.Management.factory;

import org.springframework.stereotype.Component;

import com.pharmacy.Management.models.Role;
import com.pharmacy.Management.models.User;

@Component
public class UserFactoryImpl implements UserFactory {

    @Override
    public User createUser(String username, String password, String email) {
        User user = new User();
        user.setUsername(username);
        user.setPassword(password);
        user.setEmail(email);
        user.setRole(Role.CUSTOMER); // Default role
        return user;
    }

    public User createAdmin(String username, String password, String email) {
        User user = new User();
        user.setUsername(username);
        user.setPassword(password);
        user.setEmail(email);
        user.setRole(Role.ADMIN);
        return user;
    }
}
```

3. Adapter Pattern – BillAdapter, Default BillAdapter, Bill

What it does: Allows incompatible interfaces to work together.

Where it's used: Bill is used for retrieving and storing data in db while BillAdapter is responsible for formatting the Bill details to display in admin's dashboard which is achieved by the DefaultBillAdapter.

```
public static BillAdapter getAdapter(String type) {
    if ("default".equalsIgnoreCase(type)) return new DefaultBillAdapter();
    throw new UnsupportedOperationException();
}
```

```
public class DefaultBillAdapter implements BillAdapter {
    private Bill bill;
    private static final DateTimeFormatter DATE_FORMATTER = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

    @Autowired
    private BillItemRepository billItemRepository;

    public DefaultBillAdapter() {
    }

    public DefaultBillAdapter(Bill bill) {
        this.bill = bill;
    }

    @Override
    public Bill adaptToBill() {
        return bill;
    }

    @Override
    public void adaptFromBill(Bill bill) {
        this.bill = bill;
    }
```

```java
public class Bill {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;

    private LocalDateTime billDate;
    private Double totalAmount;
    private String paymentStatus;
    private String paymentMethod;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
```

4. Singleton Pattern – Supplier

What it does: Ensures a single instance of the supplier to be maintained while providing a global access point to it.

Where it's used: New instance created as a singleton for supplier.

```
public static synchronized Supplier getInstance() {
    if (instance == null) {
        instance = new Supplier();
    }
    return instance;
}
```

Design Principles

1. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

In the code, The `CustomerController` depends on the abstract interface `CustomerFacade`, which in turn may depend on lower-level implementations. This aligns with DIP because the controller doesn't need to know or rely on how the interface logic is implemented.

CustomerController.java

src > main > java > com > pharmacy > Management > controllers > CustomerController.java > {} com.pharmacy.Management.controllers

```java
package com.pharmacy.Management.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import com.pharmacy.Management.facade.CustomerFacade;
import com.pharmacy.Management.models.Customer;

@Controller
@RequestMapping("/customers")
public class CustomerController {

    @Autowired
    private CustomerFacade customerFacade;

    @GetMapping
    public String viewCustomers(Model model) {
        model.addAttribute("customers", customerFacade.getAllCustomers());
        return "customers";
    }
```

2. Open/Closed Principle (OCP)

Software entities (classes, modules, functions) should be open for extension but closed for modification.

In the code: The billing functionality follows OCP through the `BillService` class. The bill can further be extended to include payment options also.

```java
@Transactional
public Bill createBill(Customer customer, List<BillItem> items) {
    // Save or update customer
    Customer existingCustomer = customerRepository.findByContact(customer.getContact());
    if (existingCustomer != null) {
        existingCustomer.setLastVisit(LocalDateTime.now());
        customer = customerRepository.save(existingCustomer);
    } else {
        customer.setLastVisit(LocalDateTime.now());
        customer.setTotalSpent(totalSpent:0.0);
        customer = customerRepository.save(customer);
    }

    // Create new bill
    Bill bill = new Bill();
    bill.setCustomer(customer);
    bill.setBillDate(LocalDateTime.now());
    bill.setPaymentStatus(paymentStatus:"PENDING");
    bill.setTotalAmount(totalAmount:0.0);

    double totalAmount = 0.0;
```

3. Single Responsibility Principle (SRP)

A class should handle only one functionality or one responsibility handling different functions.

In the code: Each model class handles its responsibility for example, customer handles customer login, registration and details entry. Bill handles billing of products and displaying bill details in admin dashboard.

For Customer.java

```java
public class Customer {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
private Long id;


private String name;

private String contact;

private String address;

private Integer age;

private String gender;

private LocalDateTime lastVisit;

private Double totalSpent;

private String email;


@Column(columnDefinition = "TEXT")

private String orderedDrugs;  // Will store drug names as a comma-separated string


public Long getId() {

    return id;

}


public void setId(Long id) {

    this.id = id;

}


public String getName() {

    return name;

}
```

```
public void setName(String name) {
    this.name = name;
}

public String getContact() {
    return contact;
}

public void setContact(String contact) {
    this.contact = contact;
}

public String getAddress() {
    return address;
}

public void setAddress(String address) {
    this.address = address;
}

public Integer getAge() {
    return age;
}
```

```
public void setAge(Integer age) {
    this.age = age;
}

public String getGender() {
    return gender;
}

public void setGender(String gender) {
    this.gender = gender;
}

public LocalDateTime getLastVisit() {
    return lastVisit;
}

public void setLastVisit(LocalDateTime lastVisit) {
    this.lastVisit = lastVisit;
}

public Double getTotalSpent() {
    return totalSpent;
}

public void setTotalSpent(Double totalSpent) {
```

```
        this.totalSpent = totalSpent;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getOrderedDrugs() {
        return orderedDrugs;
    }

    public void setOrderedDrugs(String orderedDrugs) {
        this.orderedDrugs = orderedDrugs;
    }
}
```

For Bill.java

```
public class Bill {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
@ManyToOne

@JoinColumn(name = "customer_id")

private Customer customer;



private LocalDateTime billDate;

private Double totalAmount;

private String paymentStatus;

private String paymentMethod;



public Long getId() {

    return id;

}



public void setId(Long id) {

    this.id = id;

}



public Customer getCustomer() {

    return customer;

}



public void setCustomer(Customer customer) {

    this.customer = customer;

}
```

```
public LocalDateTime getBillDate() {
    return billDate;
}

public void setBillDate(LocalDateTime billDate) {
    this.billDate = billDate;
}

public Double getTotalAmount() {
    return totalAmount;
}

public void setTotalAmount(Double totalAmount) {
    this.totalAmount = totalAmount;
}

public String getPaymentStatus() {
    return paymentStatus;
}

public void setPaymentStatus(String paymentStatus) {
    this.paymentStatus = paymentStatus;
}

public String getPaymentMethod() {
```

```
        return paymentMethod;

    }



    public void setPaymentMethod(String paymentMethod) {

        this.paymentMethod = paymentMethod;

    }

}
```

4. Interface Segregation Principle (ISP)

In the code: Your project uses separate interfaces for each repository. For example, the `CustomerRepository`, `SupplierRepository`, and `BillingRepository` all define only the methods relevant to their own entities.

This prevents classes from being forced to implement unused methods and keeps each repository interface focused and maintainable.

```
public interface CustomerRepository extends JpaRepository<Customer, String> {
    // only customer-related queries
}
```

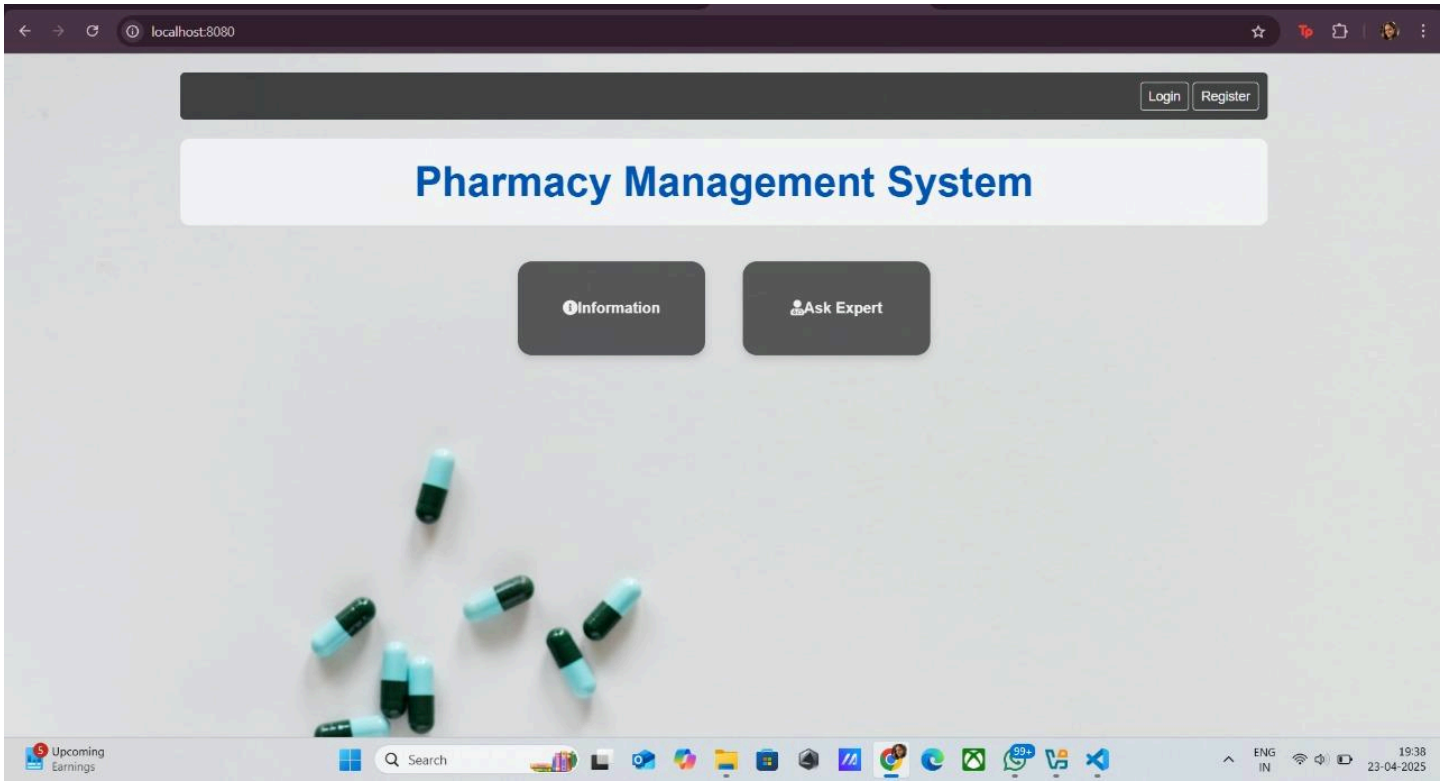
The same principle applies to the `BillAdapter` interface, where it only defines `processBill()` to avoid unnecessary complexity.

Github link to the Codebase: https://github.com/Shereen20/OOAD-Project.git

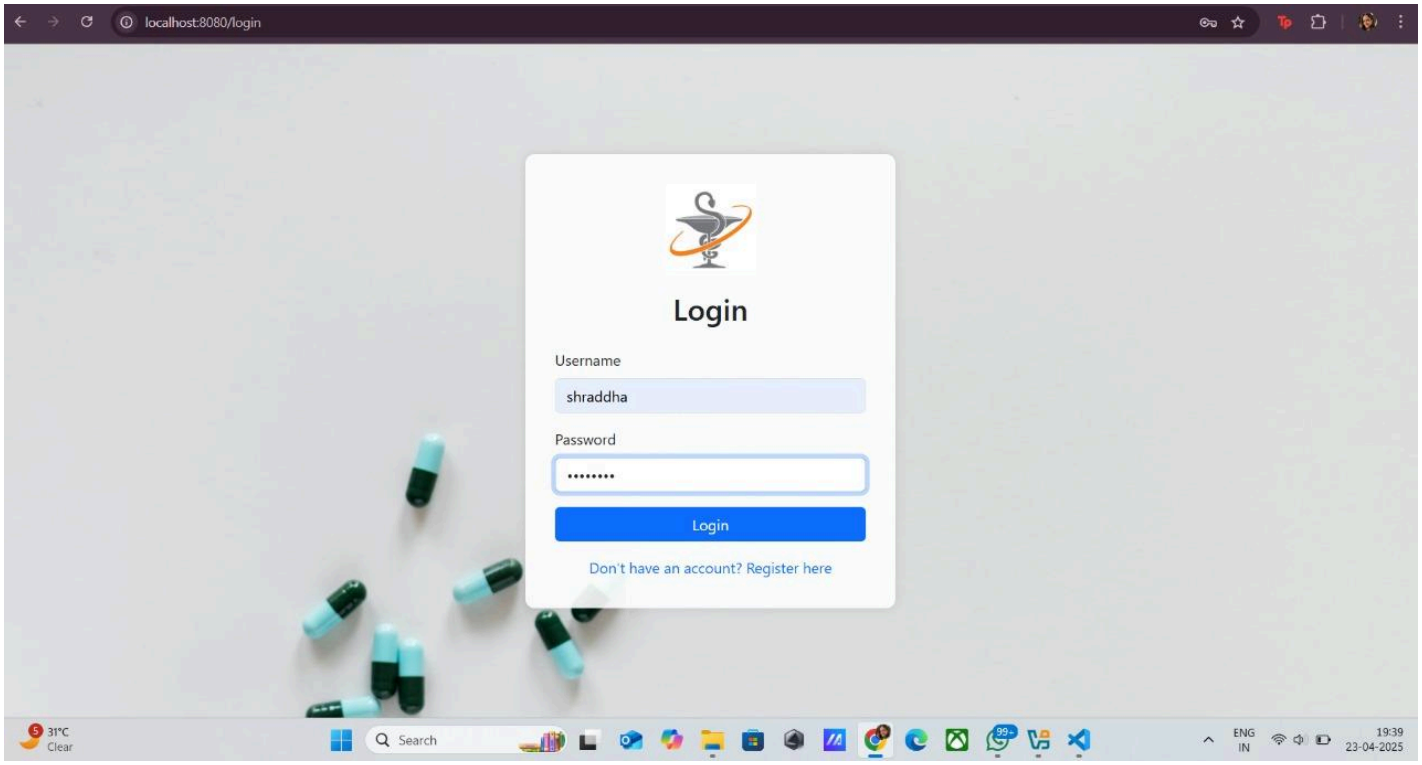
Screenshots

UI:

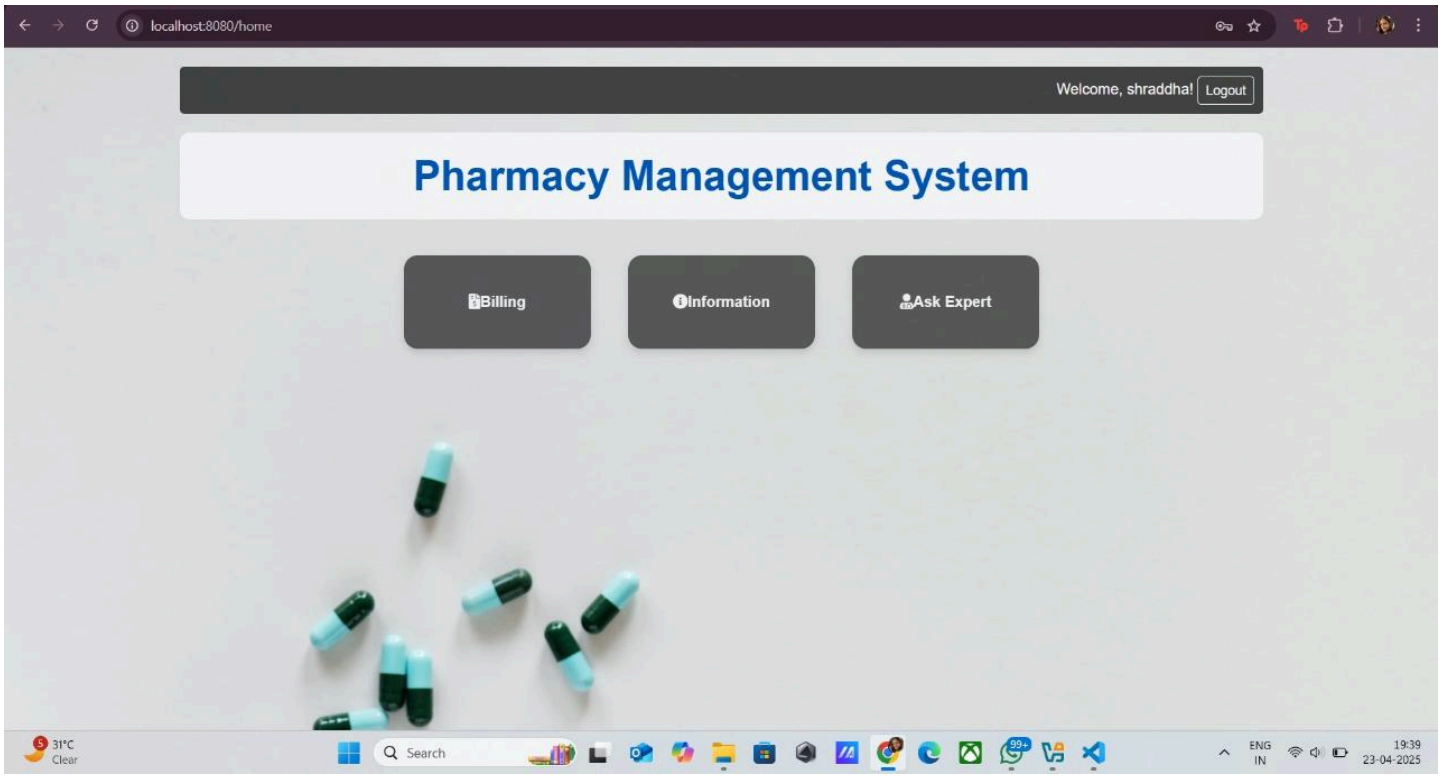
COMMON UI



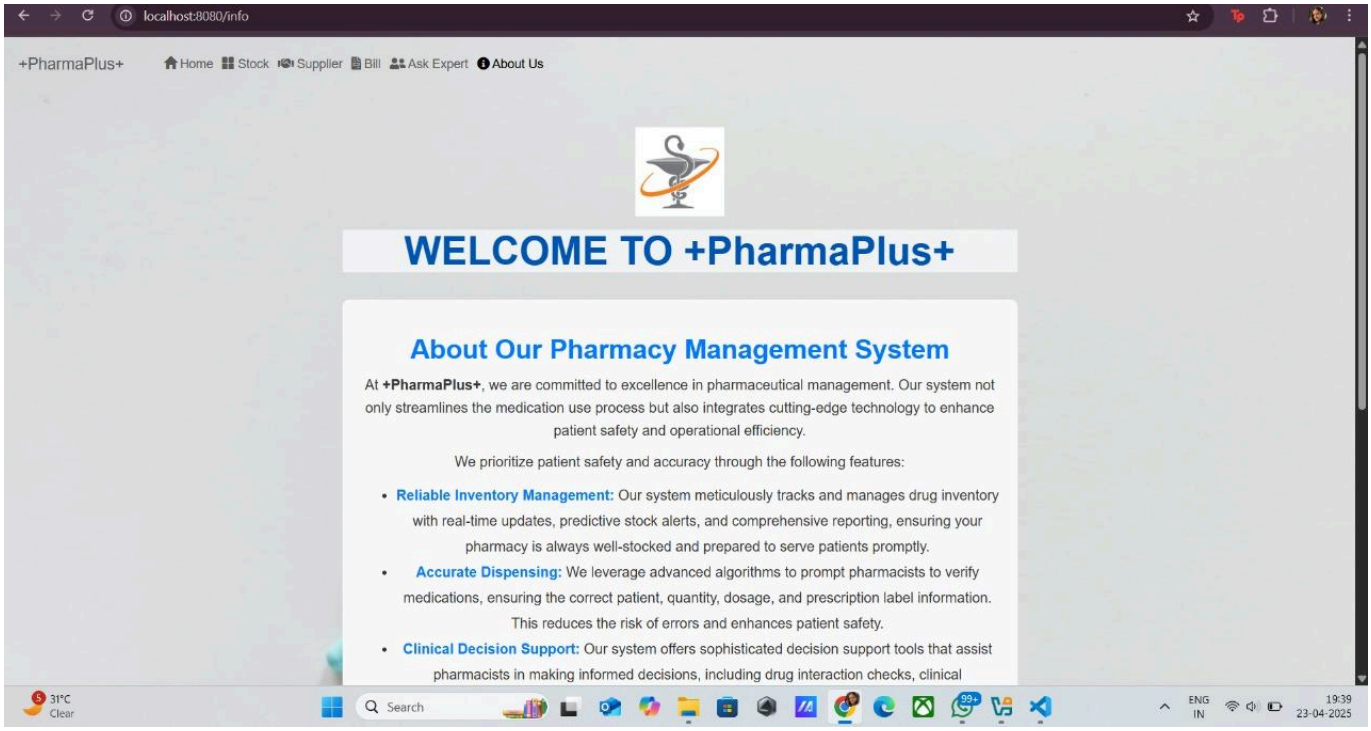
LOGIN PAGE


localhost:8080/login
Login
Username
shraddha
Password
Login
Don't have an account? Register here
31°C
Clear
Search
ENG
IN
19:39
23-04-2025

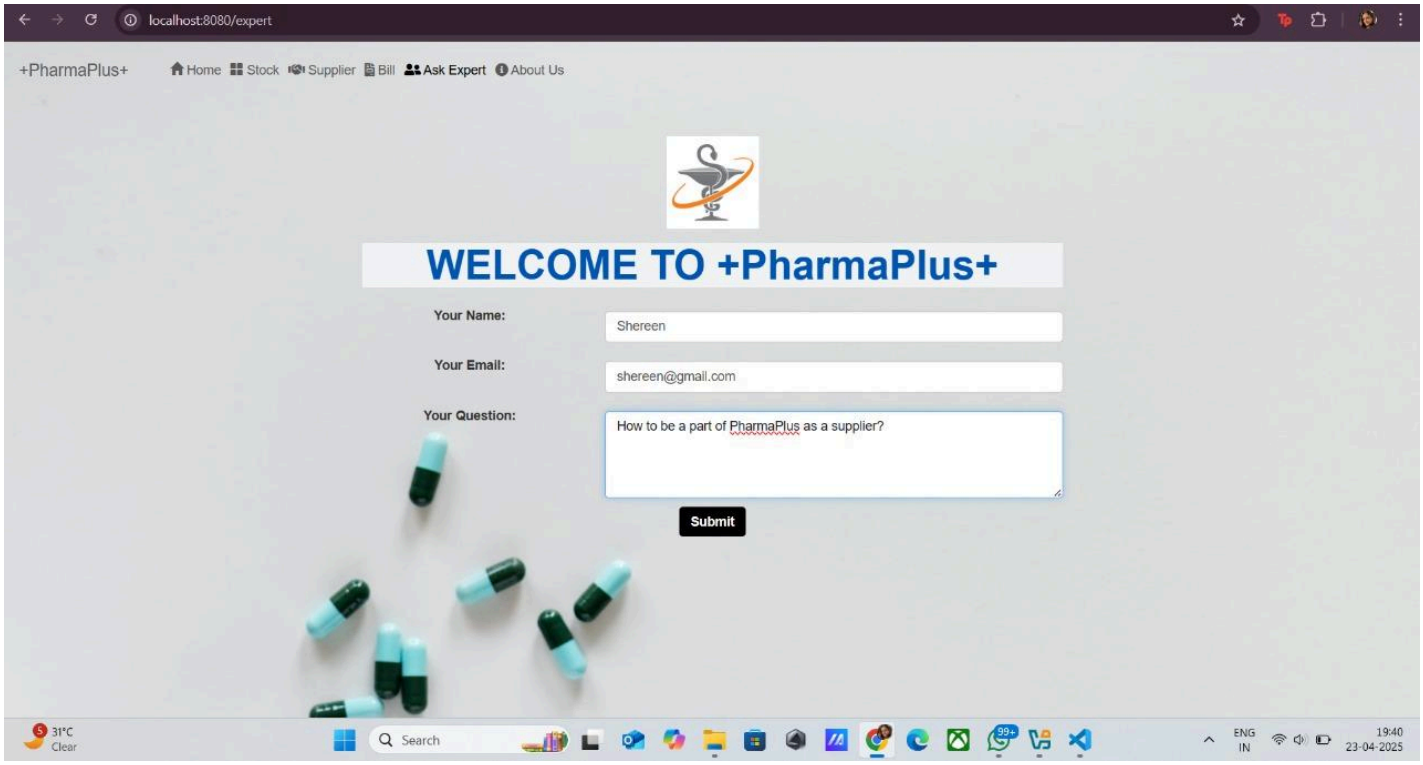
CUSTOMER HOME PAGE



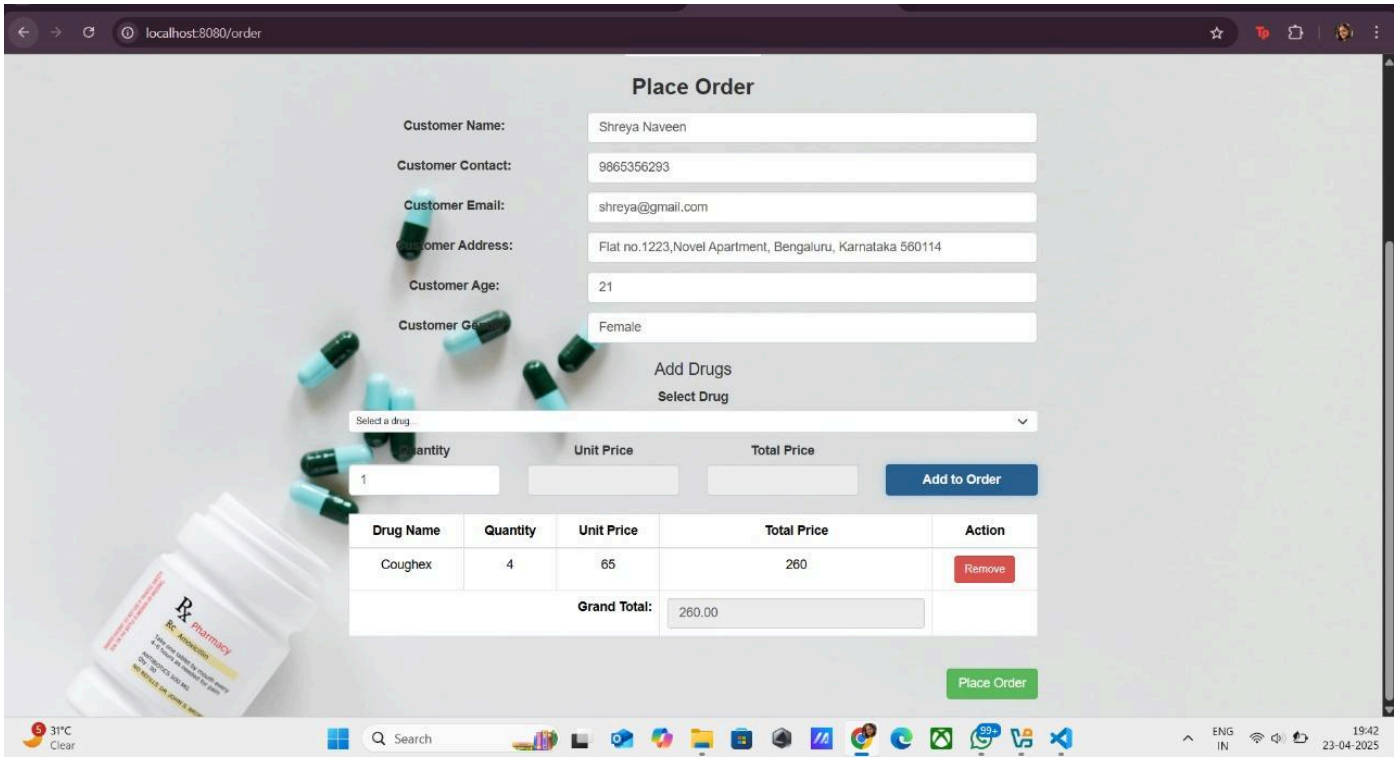
INFORMATION PAGE



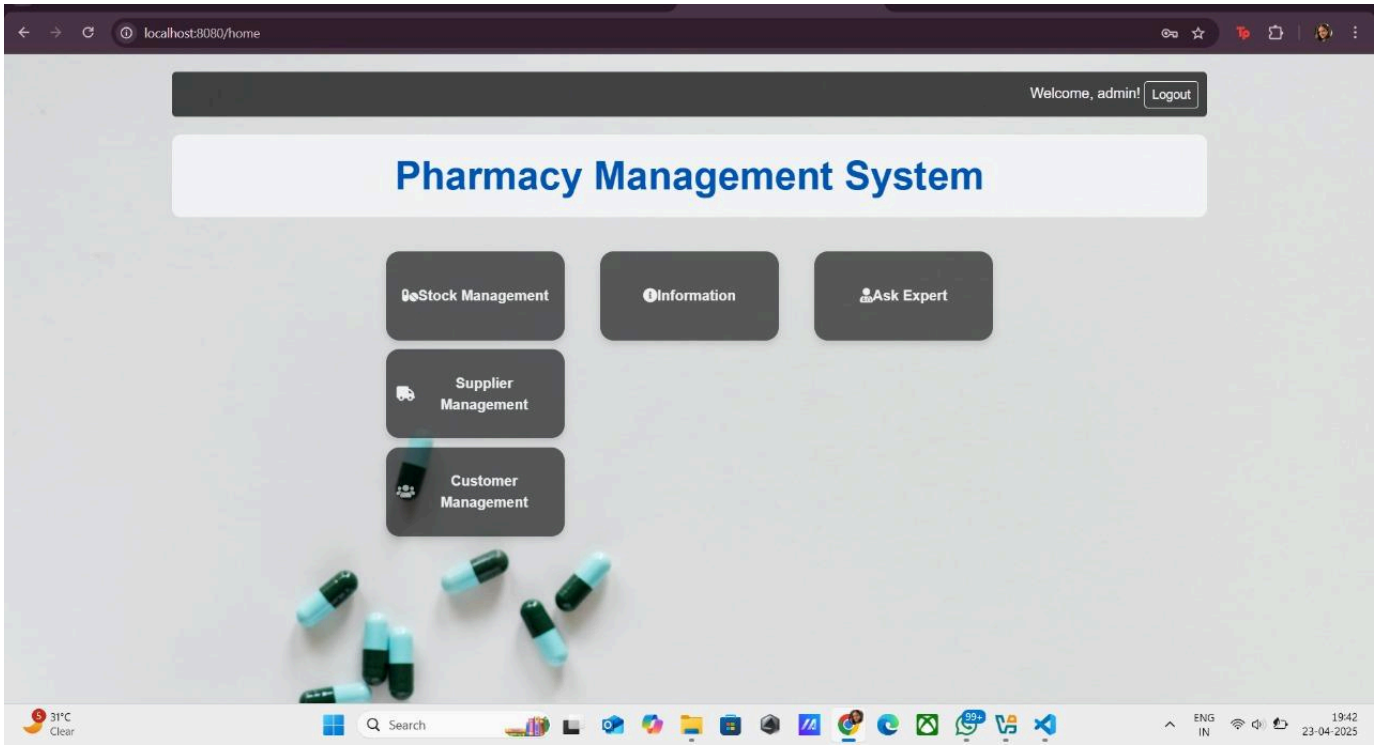
EXPERT PAGE


localhost:8080/expert
+PharmaPlus+
Home
Stock
Supplier
Bill
Ask Expert
About Us
WELCOME TO +PharmaPlus+
Your Name:
Shereen
Your Email:
shereen@gmail.com
Your Question:
How to be a part of PharmaPlus as a supplier?
Submit
31°C
Clear
Search
ENG
IN
19:40
23-04-2025

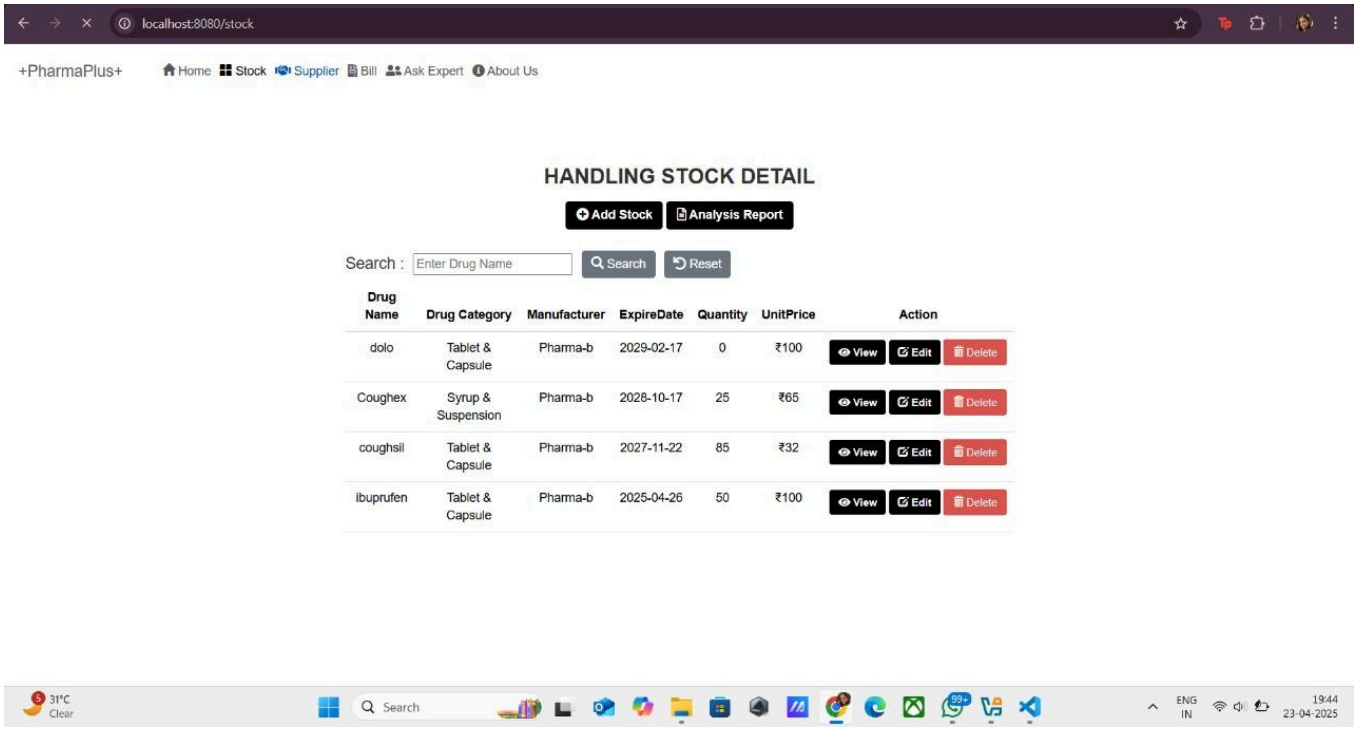
BILLING


localhost:8080/order
Place Order
Customer Name:
Shreya Naveen
Customer Contact:
9865356293
Customer Email:
shreya@gmail.com
Flat no.1223,Novel Apartment, Bengaluru, Karnataka 560114
Customer Age:
21
Female
Add Drugs
Select Drug
Select a drug...
Unit Price
Total Price
Add to Order
Drug Name
Quantity
Unit Price
Total Price
Action
Coughex
4
65
260
Remove
Grand Total:
260.00
Place Order
Search
ENG
IN
19:42
23-04-2025

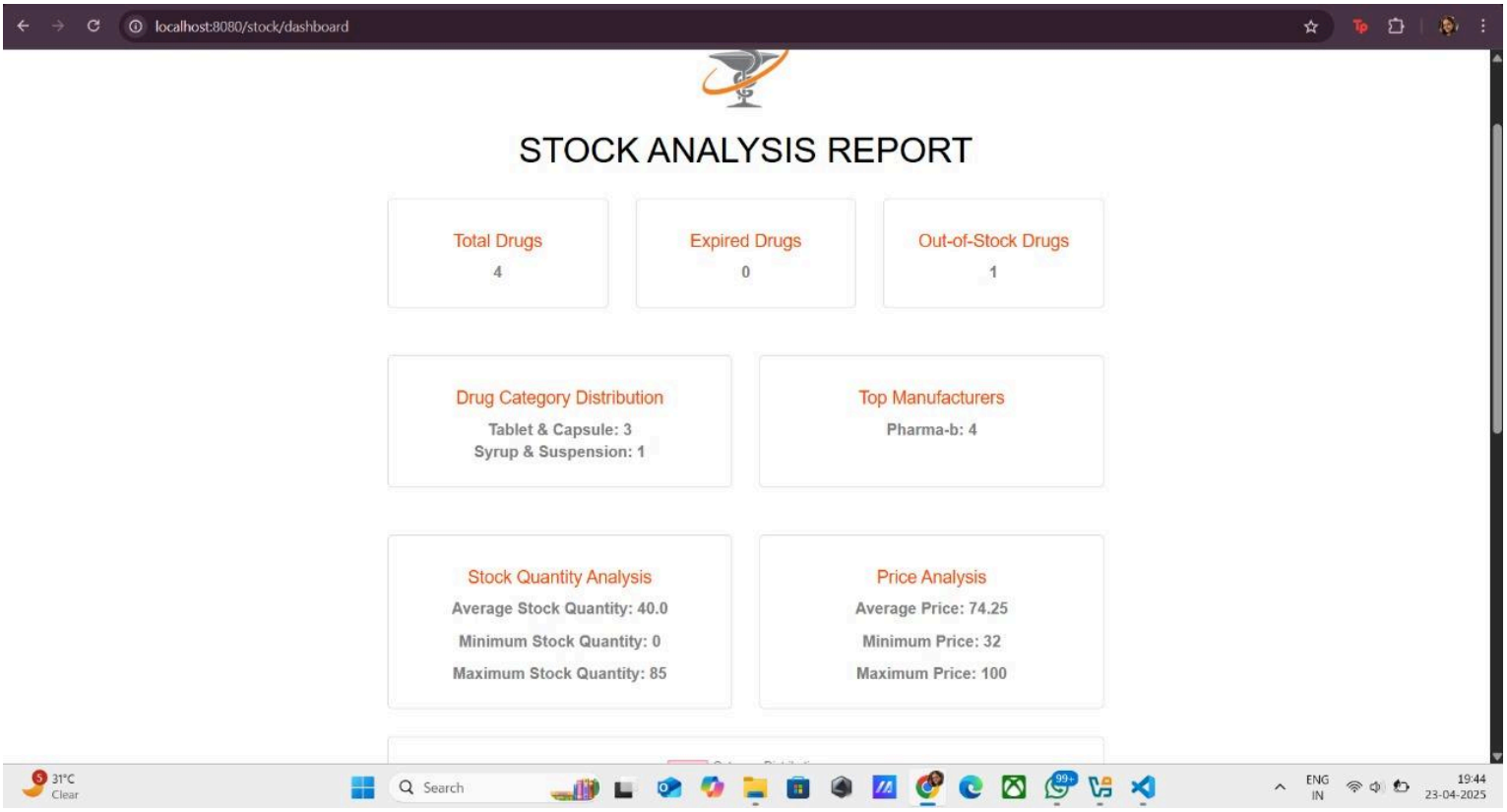
ADMIN HOME PAGE



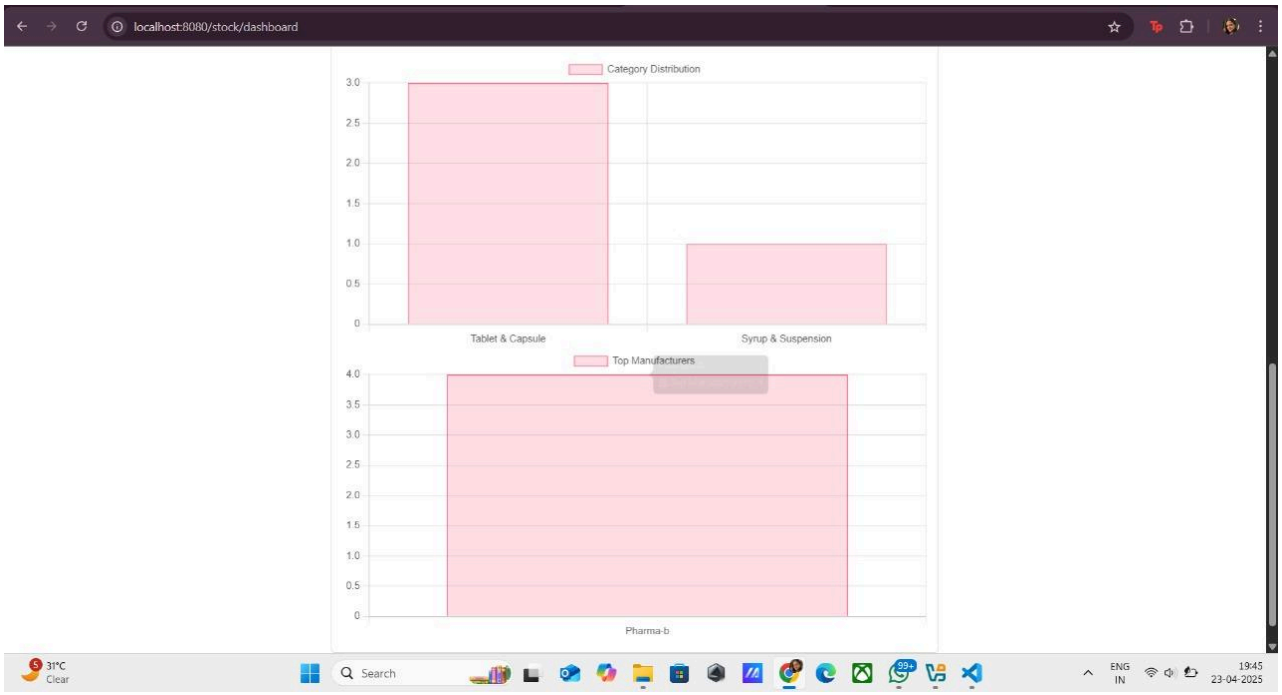
STOCK PAGE


localhost:8080/stock
+PharmaPlus+
Home
Stock
Supplier
Bill
Ask Expert
About Us
HANDLING STOCK DETAIL
Add Stock
Analysis Report
Search :
Enter Drug Name
Search
Reset
Drug Name
Drug Category
Manufacturer
ExpireDate
Quantity
UnitPrice
Action
dolo
Tablet & Capsule
Pharma-b
2029-02-17
0
₹100
View
Edit
Delete
Coughex
Syrup & Suspension
Pharma-b
2028-10-17
25
₹65
View
Edit
Delete
coughsil
Tablet & Capsule
Pharma-b
2027-11-22
85
₹32
View
Edit
Delete
ibuprufen
Tablet & Capsule
Pharma-b
2025-04-26
50
₹100
View
Edit
Delete
31°C
Clear
Search
ENG
IN
19:44
23-04-2025

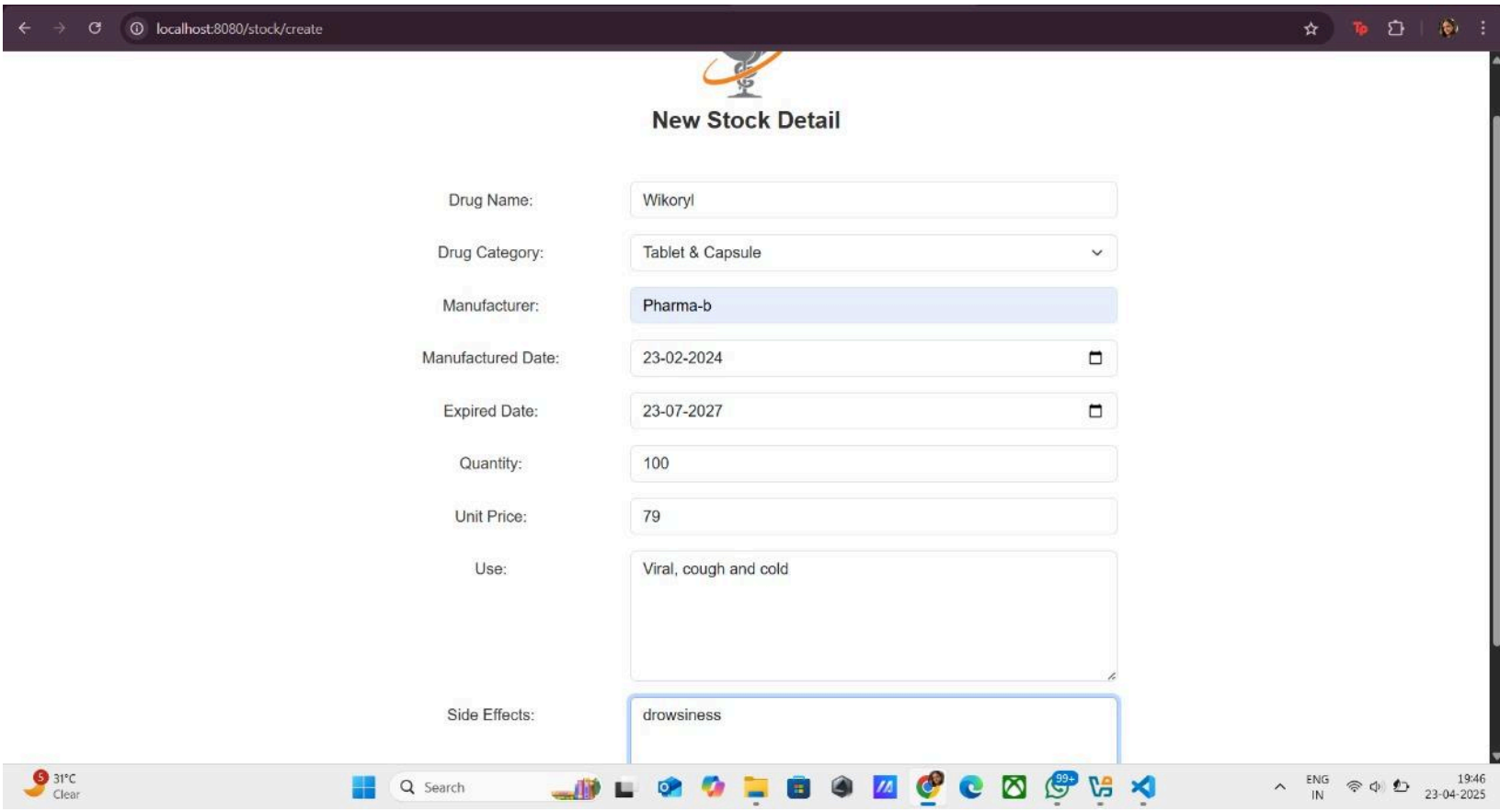
STOCK DASHBOARD



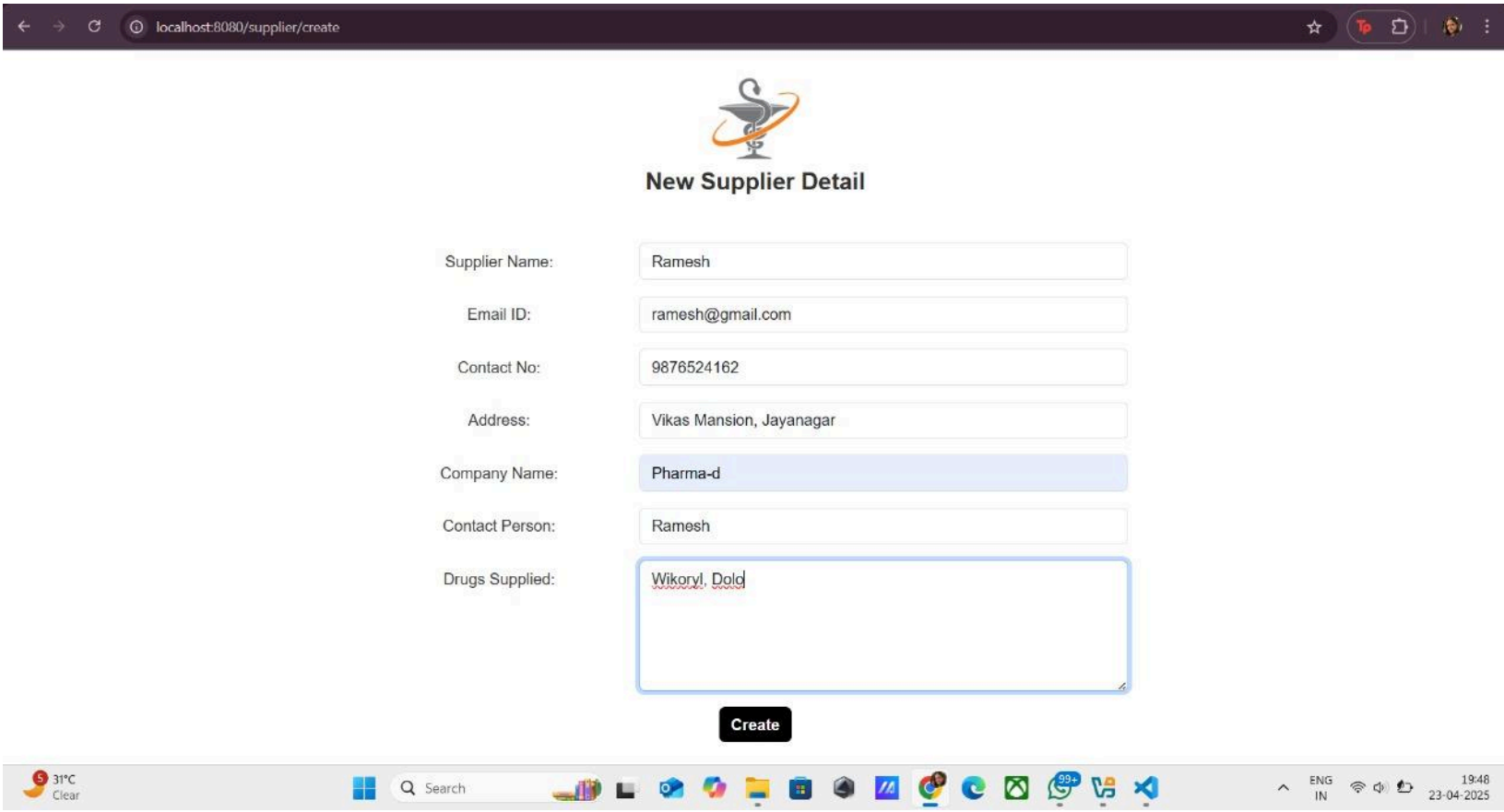
STOCK DASHBOARD



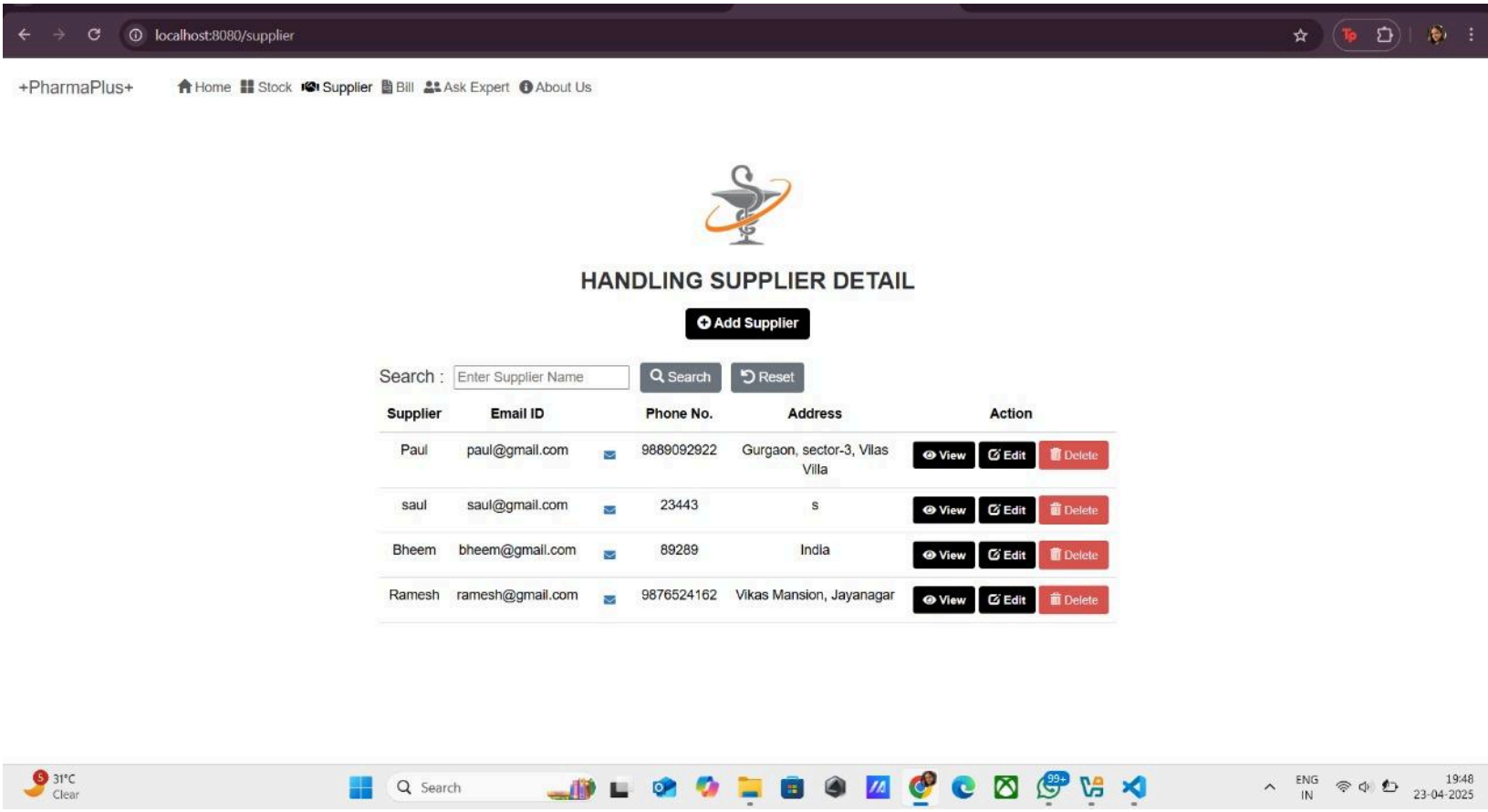
STOCK CREATION



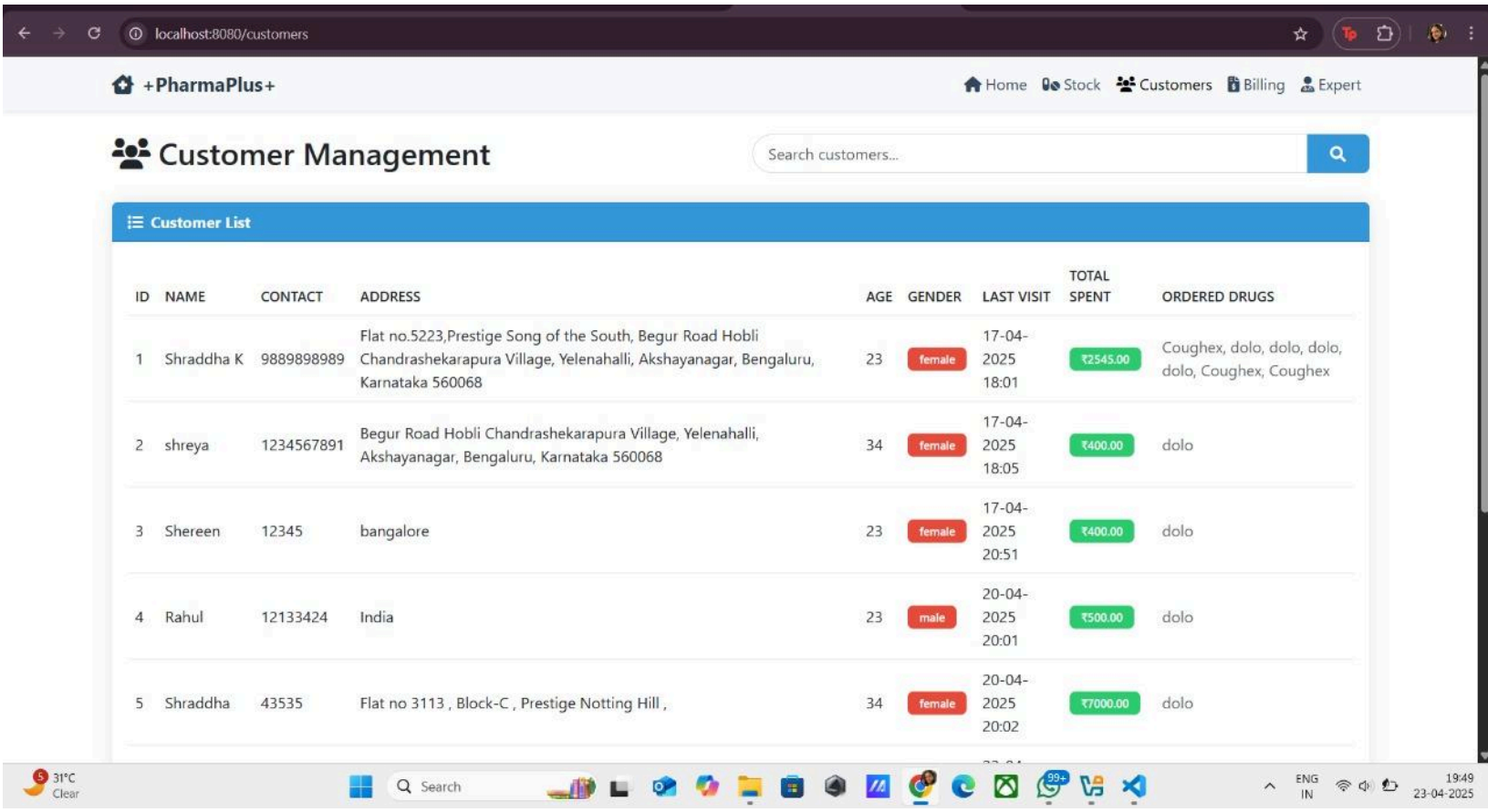
SUPPLIER ADDITION PAGE



SUPPLIER PAGE



CUSTOMER MANAGEMENT PAGE



Individual contributions of the team members:

Name	Module worked on
SHEREEN ANAND	Built a user login/signup system using Spring Security. Designed frontend pages using HTML, CSS, and Thymeleaf. Implemented Singleton Pattern.
SHRADDHA	Developed backend APIs for medicine and customer management using Spring Boot. Integrated MySQL via Spring Data JPA. Activity diagram for customer. Implemented Facade pattern.
SHREYA NAVEEN	Implemented order and billing logic in OrderController and OrderService. Created UML and use-case diagrams using Draw.io. Activity diagram for order and billing. Implemented Factory pattern.
SHREYA SRIKANT	Implemented supplier and stock logic in the respectives models and controllers. Documented API endpoints and usage in `HELP.md` and `ReadMe.md`. Activity diagram for supplier and stock. Implemented Adapter pattern.