**Software Design Project Deliverables: Online Appointment Booking System**

**1. Problem Statement and Requirements**

Business Requirements

Problem to Solve:

Allow customers to book, reschedule, or cancel appointments online.

Enable businesses to manage their schedules efficiently.

**Functionalities:**

Customer registration and login.

View available time slots and book appointments.

Notifications for appointment confirmation, reminders, and changes.

Admin dashboard for businesses to manage appointments.

**Target Users:**

Customers who want to book services.

Business owners or staff managing schedules.

**Business Goals:**

Reduce scheduling conflicts.

Improve customer satisfaction through automation.

Increase revenue by optimizing appointment slots.

Non-Functional Requirements

**Performance:**

Support up to 1,000 concurrent users.

Response time under 2 seconds for key actions.

**Security:**

User authentication and role-based access control.

Data encryption for sensitive information.

Maintainability:

Modular codebase for easy updates.

Comprehensive documentation and automated testing.

Other Requirements:

Scalability to support additional features like payments or multi-location businesses.

2. System Design Using Domain Modeling

2.1 UML Use Case Diagram

A Use Case Diagram showing actors (Customer, Admin, Notification System) and their interactions with the system.

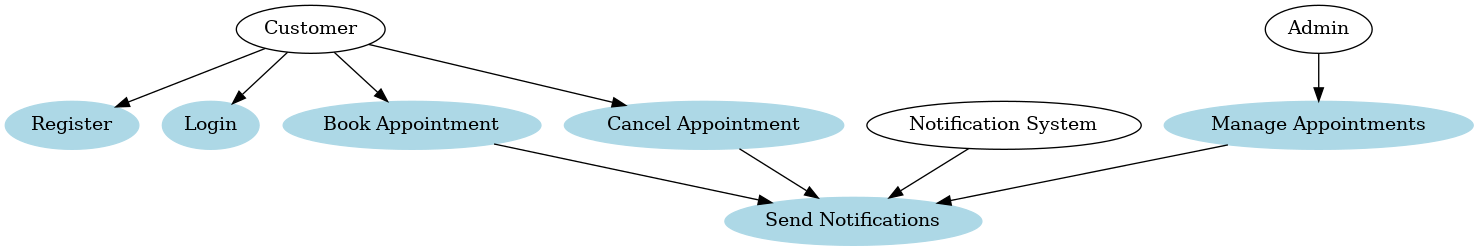
**Use Cases:**

Customer: Register, Login, Book Appointment, Cancel Appointment.

Admin: Manage Appointments.

System: Send Notifications.

Diagram: Download UML Use Case Diagram



2.2 UML Domain Model

Entities:

Customer: ID, Name, Email, Password.

Appointment: ID, Date, Time, Status, CustomerID.

Admin: ID, Role.

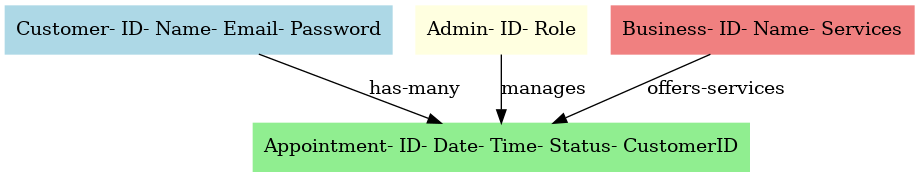
Business: ID, Name, Services.

Relationships:

Customer has many Appointments.

Admin manages Appointments.

Business offers Services.



**2.3 UML Class Diagram**

Classes:

Customer: Methods (register, login, viewAppointments).

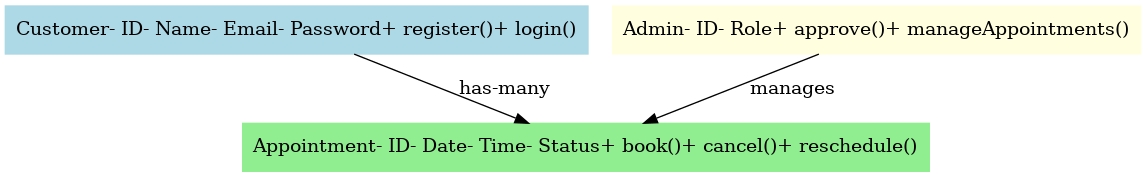
Appointment: Methods (book, cancel, reschedule).

Admin: Methods (approve, manageAppointments).

Relationships:

Customer is associated with Appointment.

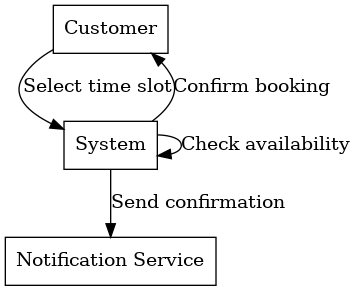
Admin has a dependency on Appointment.



2.4 UML Sequence Diagram

Use Case: Booking an Appointment

Actors: Customer, System, Notification Service.



Steps:

Customer selects a time slot.

System checks availability.

System confirms booking.

System sends a confirmation notification.

**2.5 UML State Diagram**

Appointment Lifecycle:

States: Pending, Confirmed, Completed, Cancelled.

Transitions:

Pending → Confirmed: System approves.

Confirmed → Completed: Service delivered.

Confirmed → Cancelled: Customer cancels.

2.6 UML Activity Diagram (Swimlane Diagram)

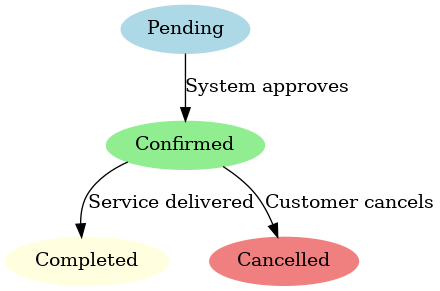
Workflow:

Swimlanes: Customer, System, Notification Service.

Steps:

Customer logs in, selects a slot, confirms booking.

System validates slot, saves appointment, notifies customer.



2.7 UML Component Diagram

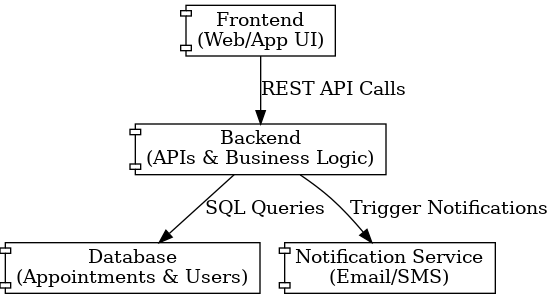
Components:

Frontend: Web/App UI.

Backend: APIs for business logic.

Database: Stores appointments and users.

Notification Service: Sends emails/SMS.



2.8 Cloud Deployment Diagram

Example on AWS:

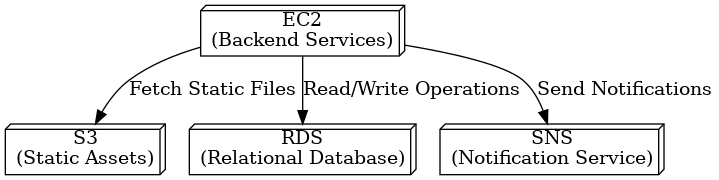
Services:

EC2: Host backend services.

S3: Store static assets.

RDS: Store relational data.

SNS: Send notifications.



**2.9 Skeleton Classes and Tables**

**Class Skeleton**:

class Customer:

def \_\_init\_\_(self, id, name, email, password):

self.id = id

self.name = name

self.email = email

self.password = password

def register(self):

# Code to register the customer

pass

def login(self):

# Code to authenticate the customer

pass

class Appointment:

def \_\_init\_\_(self, id, date, time, status, customer\_id):

self.id = id

self.date = date

self.time = time

self.status = status

self.customer\_id = customer\_id

def book(self):

# Code to book the appointment

pass

def cancel(self):

# Code to cancel the appointment

pass

def reschedule(self, new\_date, new\_time):

# Code to reschedule the appointment

pass

class Admin:

def \_\_init\_\_(self, id, role):

self.id = id

self.role = role

def manage\_appointments(self):

# Code to manage appointments

pass

**Database Table Definitions**

CREATE TABLE Customers (

ID INT PRIMARY KEY,

Name VARCHAR(255),

Email VARCHAR(255),

Password VARCHAR(255)

);

CREATE TABLE Appointments (

ID INT PRIMARY KEY,

Date DATE,

Time TIME,

Status VARCHAR(20),

CustomerID INT,

FOREIGN KEY (CustomerID) REFERENCES Customers(ID)

);

CREATE TABLE Admins (

ID INT PRIMARY KEY,

Role VARCHAR(255)

);

2.10 **Design Patterns**

**Patterns Applied:**

1. **SOLID Principles:**
   * **Single Responsibility Principle:** Each class has a specific responsibility (e.g., Customer handles customer operations, Appointment handles appointment operations).
   * **Open/Closed Principle:** Classes are open for extension but closed for modification (e.g., adding new functionalities without altering existing code).
   * **Dependency Inversion Principle:** High-level modules depend on abstractions, not on low-level modules (e.g., notification system implemented using interfaces).
2. **Observer Pattern:**
   * Used for notification services. When an appointment is booked, canceled, or rescheduled, the notification system observes and triggers appropriate alerts (e.g., SMS, email).
3. **Singleton Pattern:**
   * Ensures that the system configuration (e.g., database connections) is initialized once and shared across all components.
4. **GRASP Principles:**
   * **Controller:** The backend API acts as a controller, coordinating the flow between the frontend, database, and other services.
   * **Information Expert:** Responsibility is assigned to the class with the most information to fulfill it (e.g., Appointment class handles booking logic).
5. **Microservices Design:**
   * The system can be split into microservices (e.g., Notification Service, Appointment Service), making it scalable and easier to maintain.

**Justifications:**

* These patterns ensure scalability, maintainability, and clear separation of concerns.
* Using Observer for notifications provides flexibility to add new channels (e.g., push notifications) without modifying core logic.
* Singleton ensures resource efficiency and consistent configuration management.