University of Dhaka
Department of Computer Science and Engineering

Project: "EasyCare" a comprehensive solution of telemedicine.

Date: 26 January, 2024

Submitted by:
Md. Rokonuzaman Rokon, Roll-08
Rafi Al Sad Rifat, Roll-24
Sakib Ur Rahman, Roll-37
Md.Mahabub Hasan Mamun, Roll-43
Rhituraj Talukder, Roll-65

Submitted to:
Ismat Rahman, Asst.Professor, CSEDU
Redwan Ahmed Rizvee, Lecturer, CSEDU

Table of Contents

Abstract:

EasyCare is a telemedicine application designed to provide seamless healthcare services to users through a digital platform. It includes features like doctor categorization based on specialization, hospital listings, appointment scheduling, online prescriptions, medicine services, and blood bank services. Built using React.js for the frontend, Spring Boot for the backend, and MySQL Workbench for database management, EasyCare aims to enhance accessibility to healthcare and streamline the patient-doctor interaction process.

Introduction:

Background:

The rapid advancements in technology and the growing demand for accessible healthcare have paved the way for telemedicine as a viable solution to bridge the gap between patients and healthcare providers. The COVID-19 pandemic further emphasized the importance of remote healthcare, as traditional systems faced challenges like overcrowded facilities, long waiting times, and limited access to specialized care. EasyCare was conceived to address these issues by providing a digital platform that integrates essential healthcare services, enabling patients to access doctors, hospitals, and emergency services conveniently and efficiently from anywhere.

Problem Statement:

In today's fast-paced world, accessing timely and effective healthcare services remains a significant barrier, especially for people living in distant places or with limited mobility. Traditional healthcare systems can have long wait times, difficulty finding specialist doctors, and a lack of simplified processes for scheduling visits or acquiring medications. Delays in coordination also impede emergencies, such as the requirement for emergency service. The COVID-19 pandemic emphasized the importance of remote healthcare solutions for reducing physical interactions while providing high-quality care. EasyCare seeks to address these difficulties by offering a comprehensive telemedicine platform that integrates critical services, making healthcare more accessible, efficient, and user-friendly.

Objectives:

The main aim of the EasyCare platform is-

- To create an intuitive telemedicine platform with comprehensive features for users.
- To enhance accessibility by categorizing doctors and hospitals by specialization and location.
- To provide essential services like telemedicine video calls, online prescriptions, and blood bank services.
- To provide essential medicine service.

Scope:

The scope of the **EasyCare** project encompasses the development and deployment of a comprehensive telemedicine application that caters to a wide range of healthcare needs. Specifically, the scope includes:

1. **Target Audience:**
    - Patients seeking convenient access to healthcare services.
    - Doctors and healthcare providers looking for a digital platform to manage consultations.
    - Hospitals aim to improve patient management and outreach.
2. **Core Features:**
    - **Doctor Classification:** Displaying doctors categorized by their specialization and affiliated hospitals for easy selection.
    - **Appointment Management:** Enabling patients to schedule, reschedule, or cancel appointments based on real-time availability.
    - **Online Prescriptions:** Allowing doctors to generate and share digital prescriptions with patients.
    - **Medicine Services:** Facilitating real-time requests for emergency medicine services.
    - **Blood bank service:** Facilitating instant blood bank service.
3. **Scalability:**
    - Designed to accommodate additional features in the future, such as multilingual support, AI-based health diagnostics, and advanced analytics.
4. **Stakeholders:**
    - Includes patients, doctors, hospital administrators, and emergency service providers.

5. **Hospital Stats(Hospital statistics):**

   - This part is designed to help patients and hospital owners make informed decisions by providing insights into hospital performance across different regions. It achieves this by visualizing hospital statistics in a bar chart format.

System Design

System Architecture

The system follows a client-server architecture:

1. **Frontend**: React.js handles the user interface, providing a responsive and interactive design.
2. **Backend**: Spring Boot serves as the backend framework for handling API requests, business logic, and integration with the database.
3. **Database**: MySQL Workbench manages user data, doctor and hospital records, appointments, and prescriptions.

Technologies Used

- **Frontend**: React.js
- **Backend**: Spring Boot
- **Database**: MySQL Workbench

Database Design

The database schema includes tables for:

1. **Users**: Stores patient and doctor details.
2. **Appointments**: Tracks appointment schedules.

3. **Prescriptions**: Records consultation details and prescriptions.
4. **Blood Bank Services**: Manages emergency requests.
5. Medicine Service: Provide medicine service.

Module Description

1. **Doctor Classification**: Displays doctors by specialization and associated hospitals.
2. **Appointment Scheduling**: Enables users to book and manage appointments.
3. **Telemedicine Video Call**: Facilitates video consultations between doctors and patients.
4. **Online Prescriptions**: Generates and stores prescriptions online.
5. **Blood Bank Service:** Connects donors and patients.

Features

1. User Authentication:

The user authentication feature is implemented to ensure secure access to the application's resources. It allows users to register (sign up) for the platform, log in with valid credentials, and maintain authenticated sessions using JSON Web Tokens (JWT). This mechanism ensures that only authorized users can access protected parts of the application.

Feature implemented

User Signup:

- New users can create an account by providing a user name, email and password.
- Passwords are securely hashed before storage in the database to prevent unauthorized access in case of data breaches.

User Login:

- Existing users can log in by providing valid credentials.
- Upon successful authentication, a JWT is generated and sent to the client, which the client stores securely (e.g., in localStorage).

Token-Based Authentication:

- For every protected request, the client includes the JWT in the request header.
- The backend verifies the token, extracts user details, and grants or denies access to the requested resource.

Flow of Authentication

- The user signs up or logs in via the frontend.
- The backend validates the user credentials and generates a JWT if valid.
- The client stores the JWT and attaches it to the Authorization header for requests to protected resources.
- The backend validates the token using the JwtUtil class and processes the request if the token is valid.

2. Appointment Scheduling

The appointment scheduling feature allows patients to book appointments with their preferred doctors easily. This functionality streamlines the interaction between patients and doctors by providing an intuitive interface for selecting a doctor, scheduling a time, and confirming the booking. It eliminates manual processes, ensuring accuracy and convenience.

Feature Implemented

Patients can book appointments with doctors based on available time slots. The feature likely includes validations to check for available times and the doctor's schedule.

When an appointment is booked, canceled, or updated, notifications are sent to relevant stakeholders, such as the doctor and the patient. This ensures everyone is informed of the appointment status.

The system tracks the status of each appointment (e.g., pending, confirmed, completed, canceled). This helps both patients and doctors track the current state of the appointment.

Doctors can manage their available times for appointments. The feature likely includes a way for doctors to set working hours, as well as availability for different types of appointments (e.g., urgent or routine consultations).

Doctors can view a list of all their upcoming appointments, including details like patient information, appointment date, and status. This helps doctors manage their schedules efficiently.

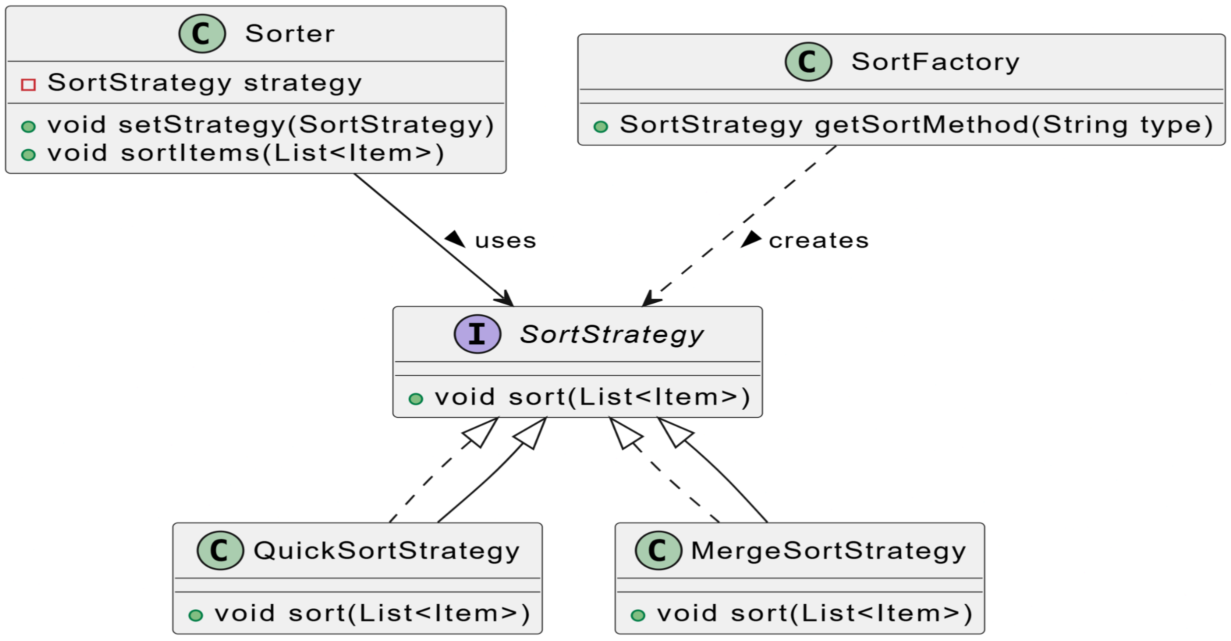
Doctors or patients can modify existing appointments. For example, the doctor might reschedule an appointment, or a patient might change the time slot. This can trigger automatic notifications or status changes.

The system likely stores a history of past appointments, allowing doctors and patients to review their previous consultations and treatments.

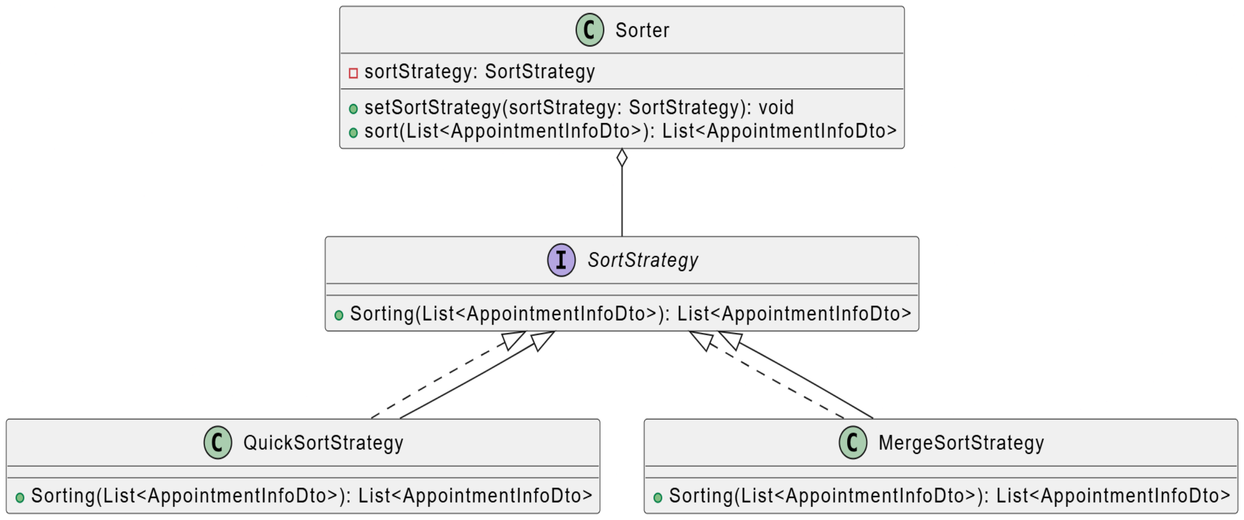
The Appointment Doctor feature is likely integrated with other modules like Prescription Management (for creating prescriptions after appointments) and Patient Management (for accessing patient details during consultations).

Pattern

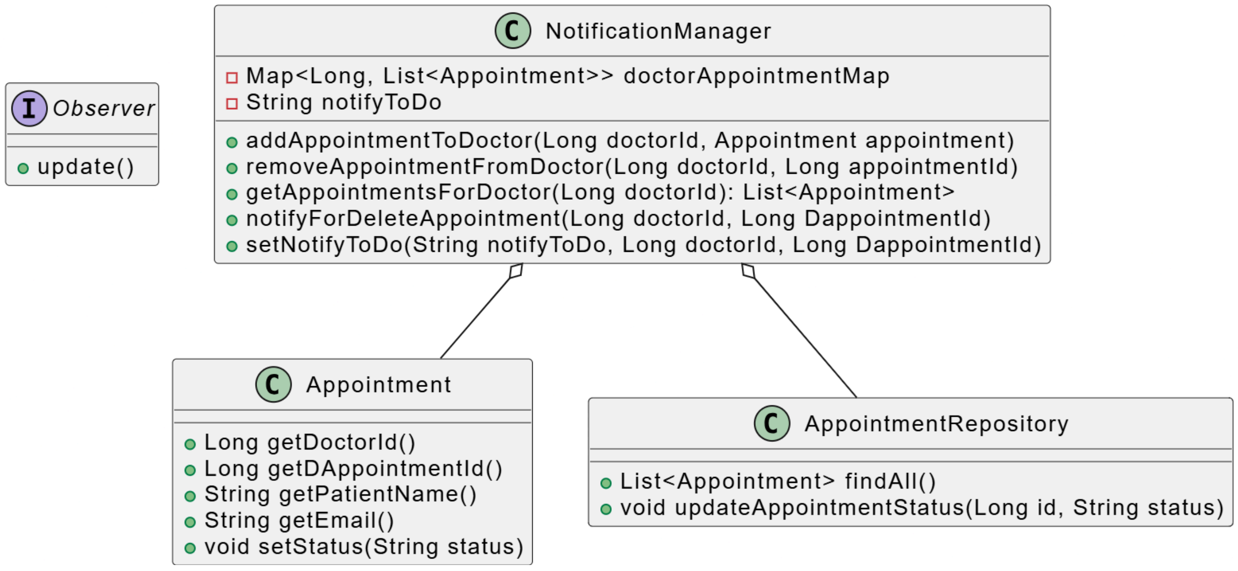
1. **Factory Pattern**: The SortFactory class is responsible for creating instances of the SortStrategy based on the input string . The pattern centralizes the creation logic in a factory class, hiding the details of object instantiation from the client.The factory can decide which class to instantiate based on runtime conditions or parameters.Adding new types of objects only requires extending the factory without changing client code.



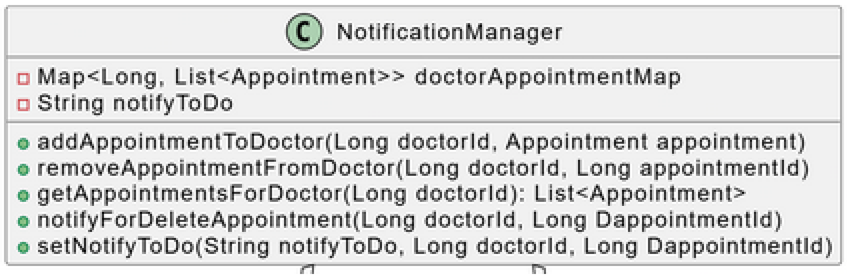
2. **Strategy Pattern:** This diagram illustrates the Strategy Design Pattern applied to sorting appointments. The Sorter class delegates the sorting task to a SortStrategy, allowing a dynamic selection of sorting algorithms. The SortStrategy is an interface that defines a common method of Sorting, which takes a list of AppointmentInfoDto and returns a sorted list. Concrete strategies like QuickSortStrategy and MergeSortStrategy implement the SortStrategy interface, encapsulating different sorting algorithms. By setting the SortStrategy in Sorter at runtime, the system can flexibly switch between sorting techniques without altering the core logic, promoting code reusability and scalability.


Sorter
sortStrategy: SortStrategy
setSortStrategy(sortStrategy: SortStrategy): void
sort(List<AppointmentInfoDto>): List<AppointmentInfoDto>
SortStrategy
Sorting(List<AppointmentInfoDto>): List<AppointmentInfoDto>
QuickSortStrategy
Sorting(List<AppointmentInfoDto>): List<AppointmentInfoDto>
MergeSortStrategy
Sorting(List<AppointmentInfoDto>): List<AppointmentInfoDto>

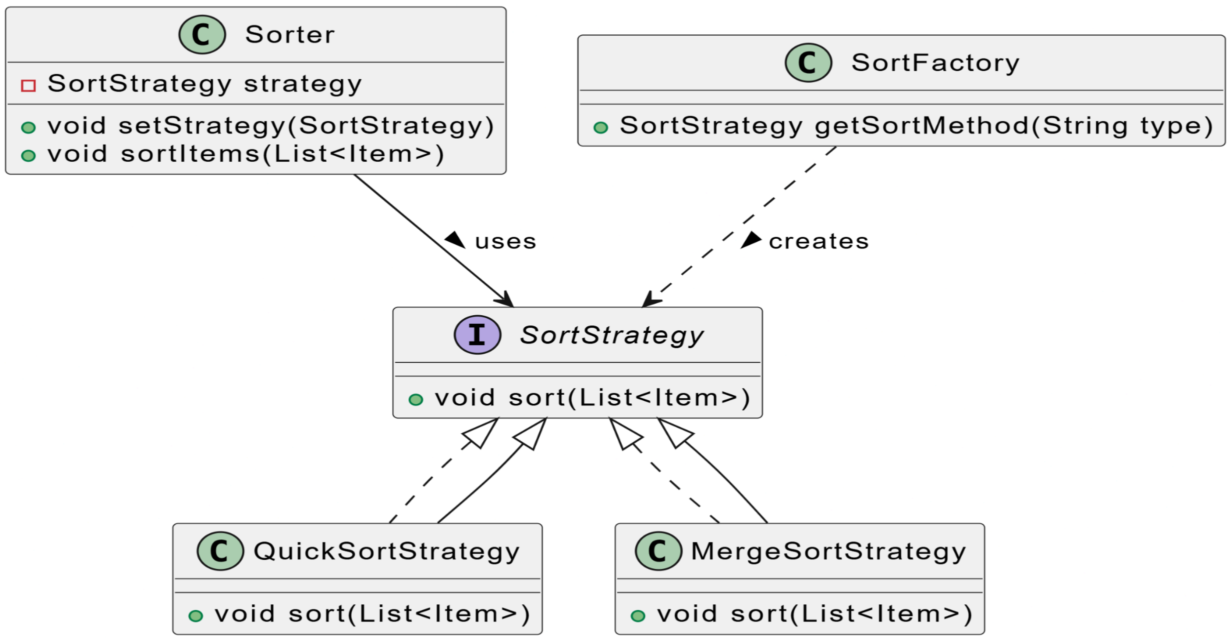
3. **Observer Pattern:** The Observer Pattern is used when notifications or reminders need to be sent to doctors or patients about their appointments. The NotificationManager can act as an observer that listens for changes in appointments (e.g., appointment cancellations or status updates) and notifies the relevant stakeholders.


Observer
update()
NotificationManager
Map<Long, List<Appointment>> doctorAppointmentMap
String notifyToDo
addAppointmentToDoctor(Long doctorId, Appointment appointment)
removeAppointmentFromDoctor(Long doctorId, Long appointmentId)
getAppointmentsForDoctor(Long doctorId): List<Appointment>
notifyForDeleteAppointment(Long doctorId, Long DappointmentId)
setNotifyToDo(String notifyToDo, Long doctorId, Long DappointmentId)
Appointment
Long getDoctorId()
Long getDAppointmentId()
String getPatientName()
String getEmail()
void setStatus(String status)
AppointmentRepository
List<Appointment> findAll()
void updateAppointmentStatus(Long id, String status)

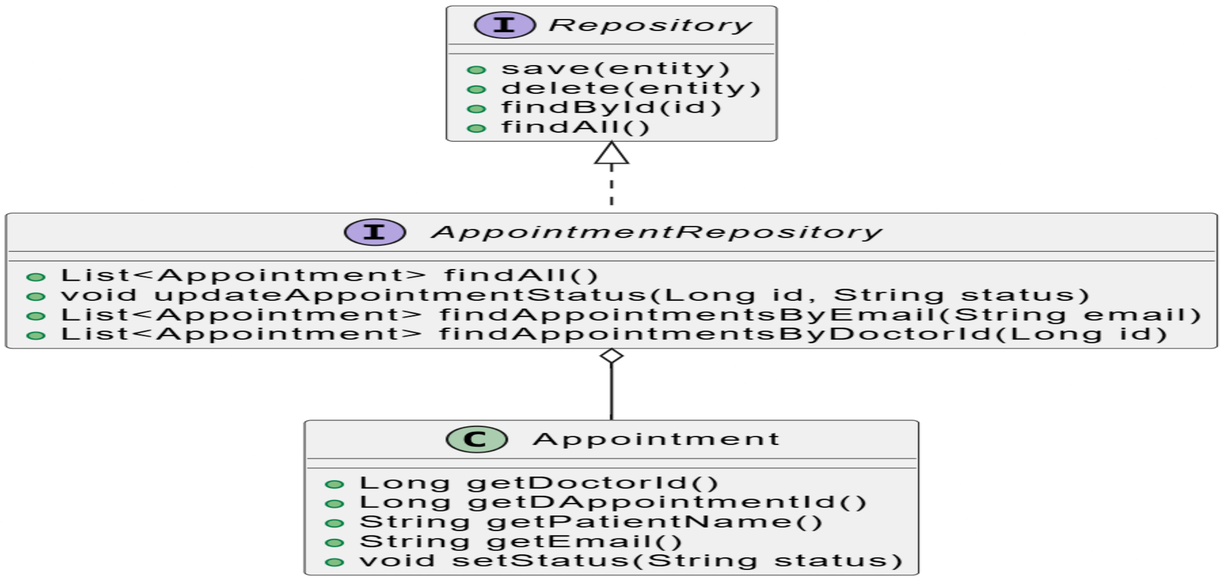
4. **Singleton Pattern:** The Singleton Pattern is used in the NotificationManager to ensure that only one instance of the notification logic exists within the application. This pattern is useful for centralized notification handling, ensuring that the same instance of the NotificationManager is used throughout the lifecycle of the application.


NotificationManager
Map<Long, List<Appointment>> doctorAppointmentMap
String notifyToDo
addAppointmentToDoctor(Long doctorId, Appointment appointment)
removeAppointmentFromDoctor(Long doctorId, Long appointmentId)
getAppointmentsForDoctor(Long doctorId): List<Appointment>
notifyForDeleteAppointment(Long doctorId, Long DappointmentId)
setNotifyToDo(String notifyToDo, Long doctorId, Long DappointmentId)

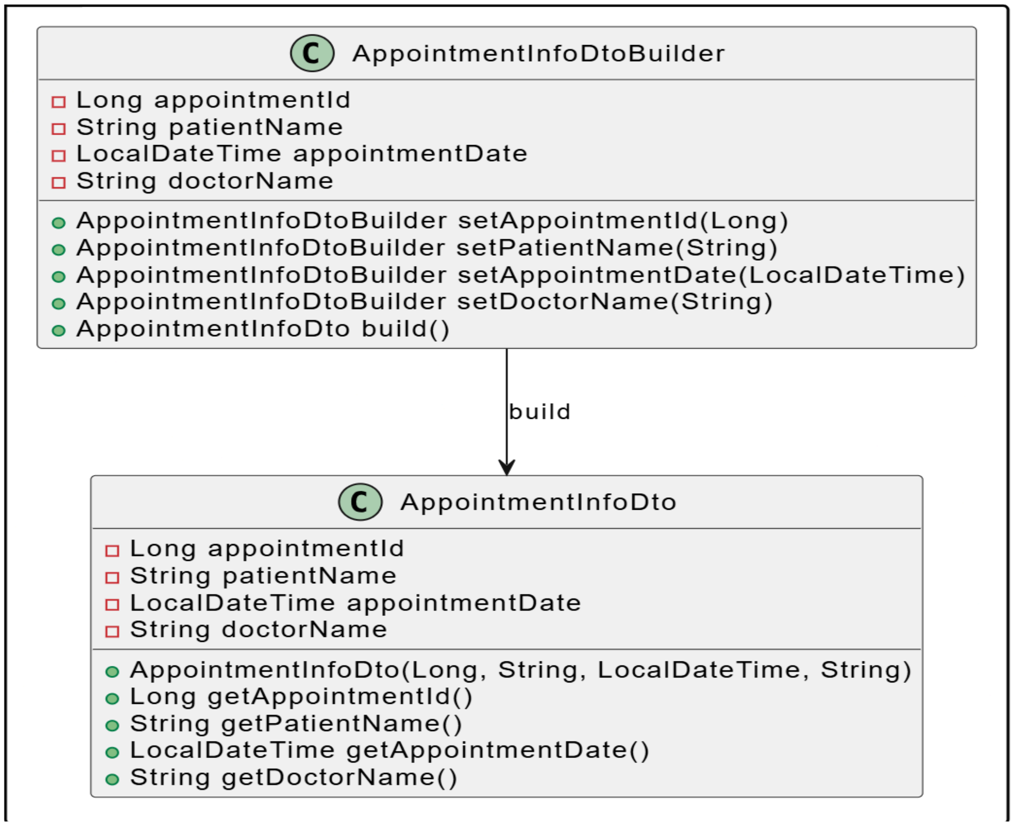
Object Creation of SortFactory Class also follows a singleton design pattern .



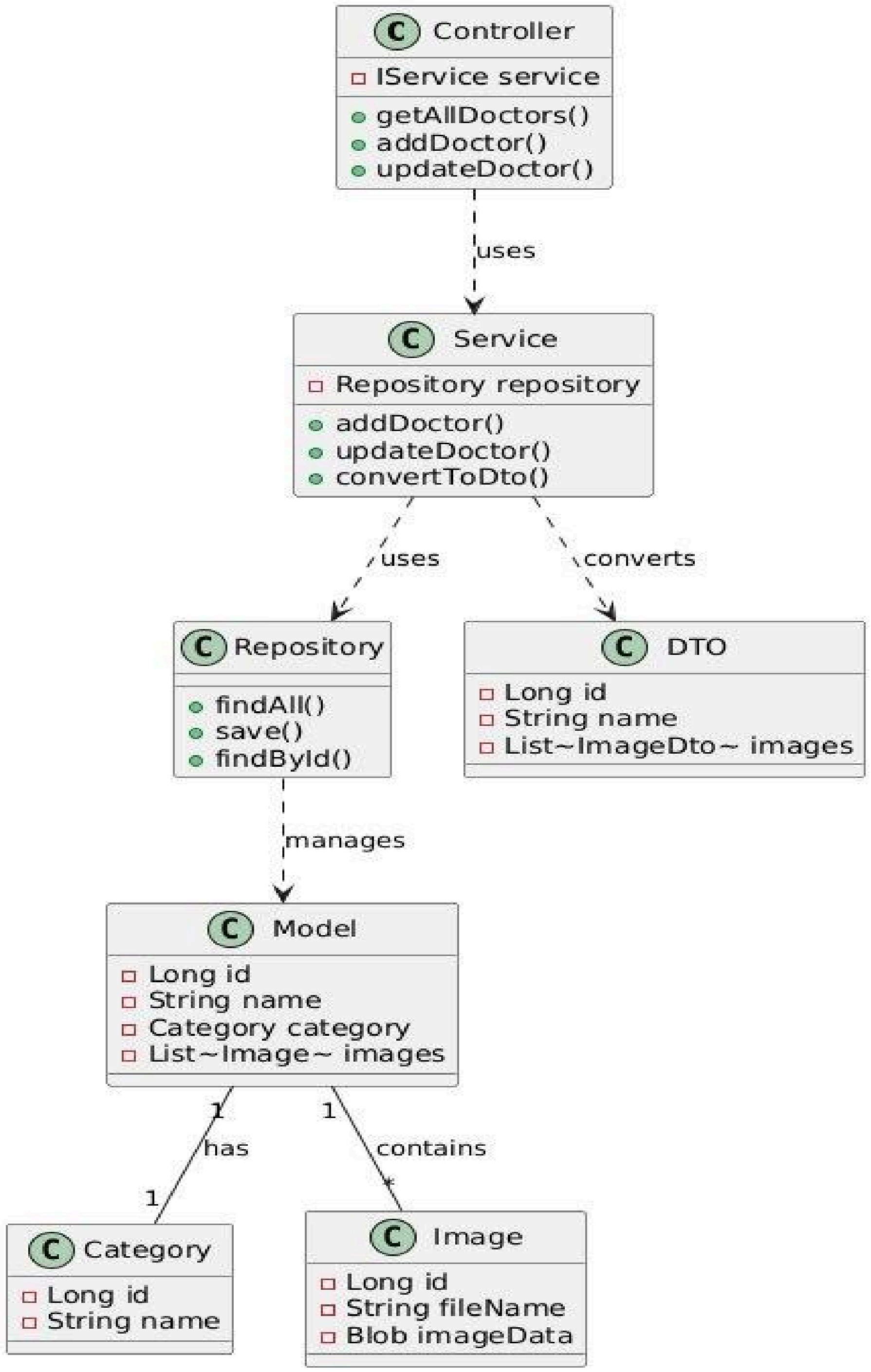
5. **Repository Pattern:** The Repository Pattern is used extensively in the data access layer. The AppointmentRepository interacts with the database, abstracting the actual data storage mechanism from the rest of the application. This pattern helps in querying appointments and updating their status without directly dealing with the underlying database.


Repository
save(entity)
delete(entity)
findById(id)
findAll()
AppointmentRepository
List<Appointment> findAll()
void updateAppointmentStatus(Long id, String status)
List<Appointment> findAppointmentsByEmail(String email)
List<Appointment> findAppointmentsByDoctorId(Long id)
Appointment
Long getDoctorId()
Long getDAppointmentId()
String getPatientName()
String getEmail()
void setStatus(String status)

6. **Builder Pattern:** The Builder Pattern is used in the Appointment Feature to simplify the creation of complex `Appointment` objects with multiple attributes, such as doctor details, patient details, appointment date, time, status, and optional notes. It enables step-by-step construction of these objects using a fluent API, avoiding the need for a cumbersome constructor with numerous parameters. This approach improves code readability, allows for flexible and customizable appointment creation, and ensures the immutability of the final object once built. By encapsulating the construction logic, the Builder Pattern makes the process more maintainable and adaptable to future changes.


AppointmentInfoDtoBuilder
Long appointmentId
String patientName
LocalDateTime appointmentDate
String doctorName
AppointmentInfoDtoBuilder setAppointmentId(Long)
AppointmentInfoDtoBuilder setPatientName(String)
AppointmentInfoDtoBuilder setAppointmentDate(LocalDateTime)
AppointmentInfoDtoBuilder setDoctorName(String)
AppointmentInfoDto build()
build
AppointmentInfoDto
Long appointmentId
String patientName
LocalDateTime appointmentDate
String doctorName
AppointmentInfoDto(Long, String, LocalDateTime, String)
Long getAppointmentId()
String getPatientName()
LocalDateTime getAppointmentDate()
String getDoctorName()

Doctor appointment full structure



3. Blood donation system

The blood donation system is designed to allow users to manage their personal information, post urgent blood requests, and interact with the platform in a user-friendly manner. The system facilitates the management of user data, including blood group, location, contact details, and notifications for urgent blood requests. It also integrates with Google Maps to provide real-time location information about blood banks.

Features Implemented

Users can add, edit, and delete their personal information, including blood group, location, phone number, and email. This is essential for maintaining an up-to-date profile and facilitating the donation process.

Users can view their added information, allowing them to check and update their profile details.

Users can post a request for urgent blood donations, specifying details about their requirements. This feature allows users to urgently request blood donations when needed.

Users can post an urgent blood request by filling out the following fields:

Name: The name of the person requiring the blood.

Blood Group: The required blood group (e.g., A+, B-, O+, etc.).

Location: The location where the blood is needed (e.g., city, hospital).

Phone Number: The contact number for further communication.

Email: The contact email for communication or updates.

Required Date: The date on which the blood is required.

Required Time: The time by which the blood is required.

Quantity (in bags): The number of blood bags needed.

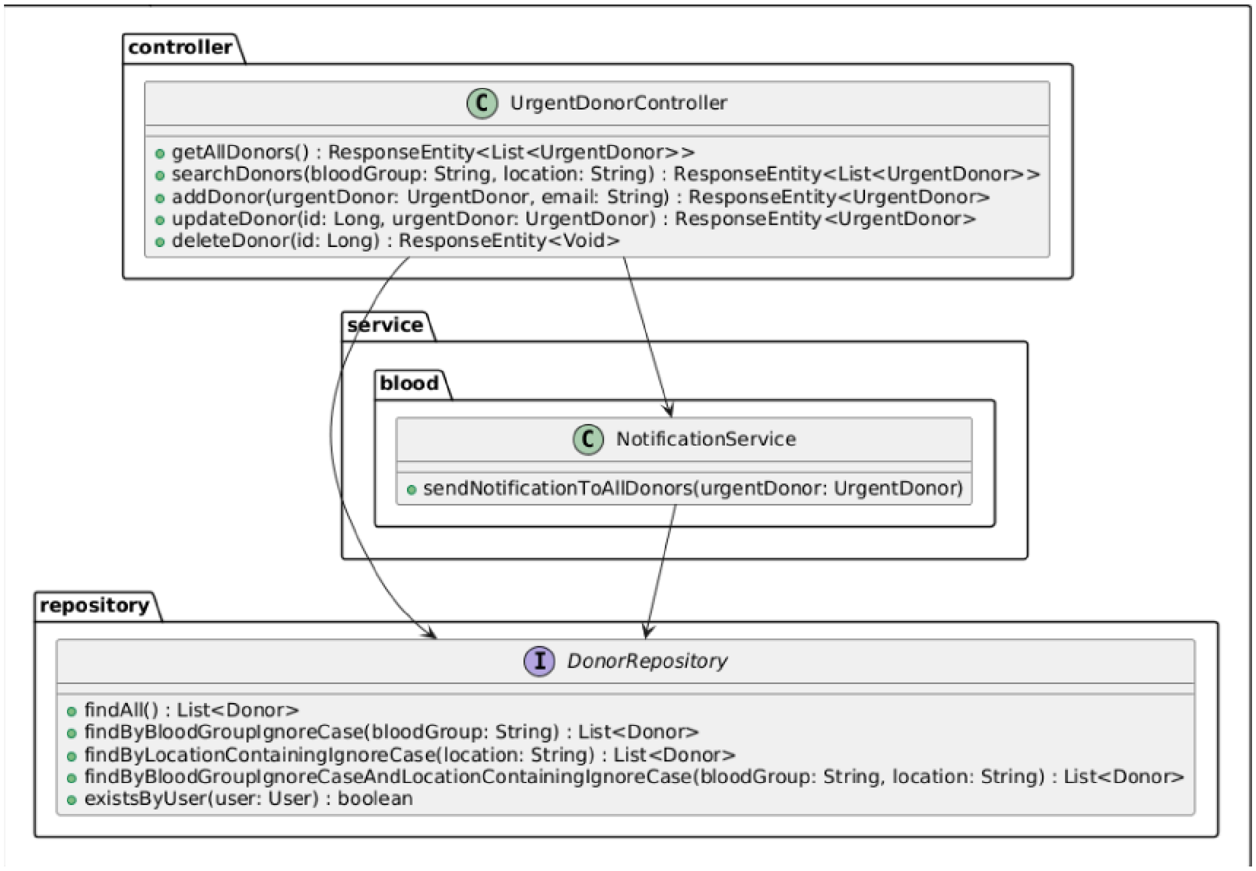
When an urgent blood request is posted, a notification is sent to all the donors who are interested in donating blood. This helps spread the word to potential donors quickly and efficiently.

Users can delete their urgent blood requests once their need is fulfilled or they no longer require the donation.

A button on the page allows users to view blood bank locations on a Google Map. Clicking the button displays nearby blood banks to help users find the closest facility for donation or for receiving blood.

Observer Pattern (for Notifications):

The Observer Pattern is used to notify users when an urgent blood request is posted. The system sends notifications to all users who wish to donate blood, and these users are notified in real-time when a new request is posted. The user's email, stored in localStorage, acts as the key to fetching specific notifications from the server.


controller
UrgentDonorController
getAllDonors() : ResponseEntity<List<UrgentDonor>>
searchDonors(bloodGroup: String, location: String) : ResponseEntity<List<UrgentDonor>>
addDonor(urgentDonor: UrgentDonor, email: String) : ResponseEntity<UrgentDonor>
updateDonor(id: Long, urgentDonor: UrgentDonor) : ResponseEntity<UrgentDonor>
deleteDonor(id: Long) : ResponseEntity<Void>
service
blood
NotificationService
sendNotificationToAllDonors(urgentDonor: UrgentDonor)
repository
DonorRepository
findAll() : List<Donor>
findByBloodGroupIgnoreCase(bloodGroup: String) : List<Donor>
findByLocationContainingIgnoreCase(location: String) : List<Donor>
findByBloodGroupIgnoreCaseAndLocationContainingIgnoreCase(bloodGroup: String, location: String) : List<Donor>
existsByUser(user: User) : boolean

4. Medicine Service Feature

In the Medicine Service feature, several key functionalities are implemented, focusing primarily on managing and processing medicine-related data.

Feature Implemented

The company can add medicine by its name, brand, price, description, inventory and most importantly the medicine category.

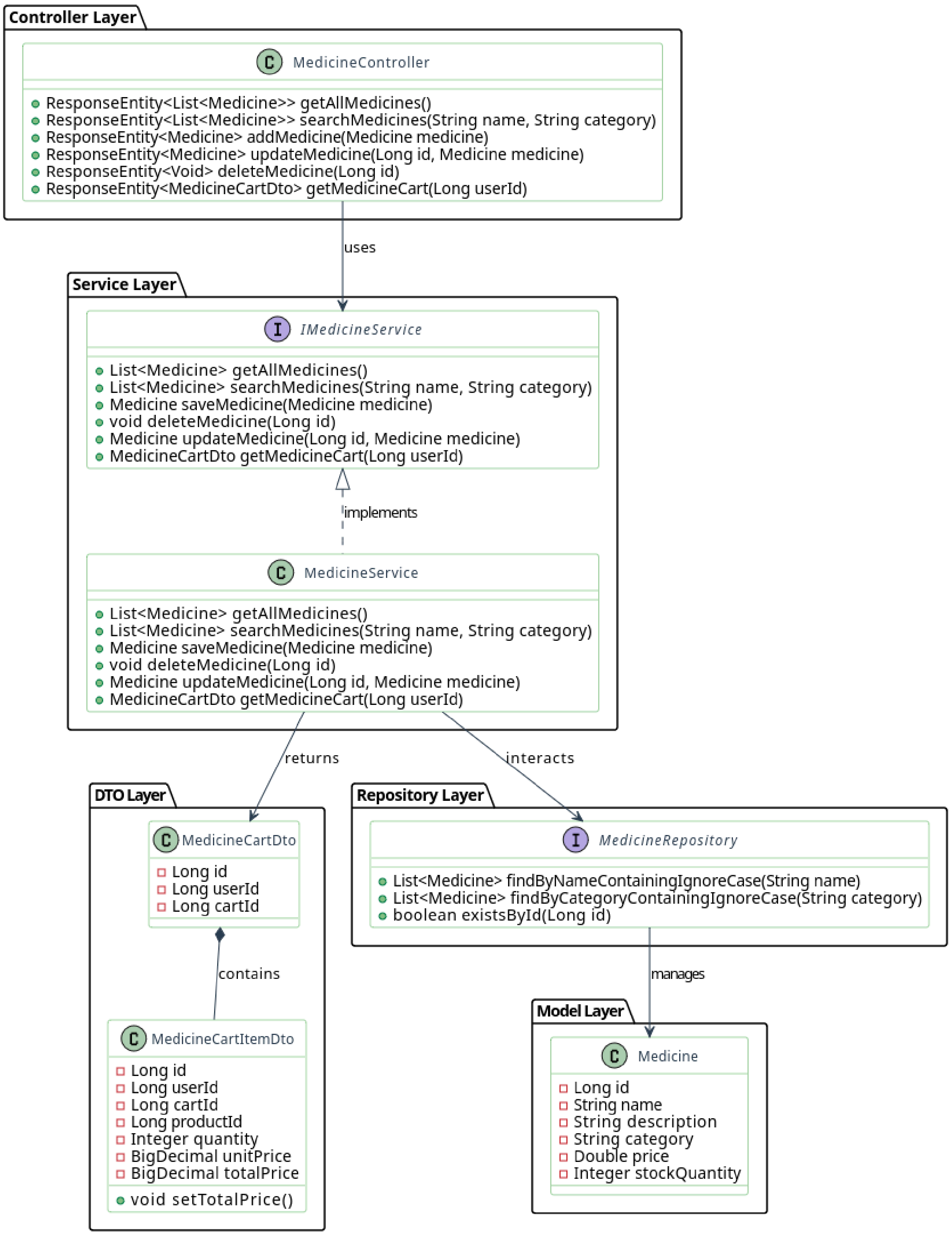
Patients can search medicine by the name and category dynamically.

Patients can take certain medicines from the list and the price will be calculated automatically.

In the cart the user can see the total price of all the medicines.

Pattern:

1. **Repository Pattern:** The Repository Pattern is used in the Medicine feature to abstract the database operations for the Cart and CartItem entities. It encapsulates the logic required to access the data source and provides a clean API for the service layer to perform CRUD operations without directly interacting with the database. By using repositories like CartRepository and CartItemRepository, the system ensures that data access logic is centralized, reusable, and decoupled from the business logic, promoting maintainability and testability.
2. **Singleton Pattern:** In the Medicine feature, the Singleton Pattern can be observed in the design of service classes like MedicineCartService and MedicineCartItemService. These classes are typically managed as singletons by the Spring Framework, meaning only one instance of each service is created and shared across the application. Spring handles this by default through the @Service annotation, ensuring that a single instance of the service exists in the application's context. This pattern helps optimize memory usage and ensures consistent behavior, as the service's state is maintained throughout the application lifecycle, promoting reusability and reducing overhead.


Controller Layer
MedicineController
ResponseEntity<List<Medicine>> getAllMedicines()
ResponseEntity<List<Medicine>> searchMedicines(String name, String category)
ResponseEntity<Medicine> addMedicine(Medicine medicine)
ResponseEntity<Medicine> updateMedicine(Long id, Medicine medicine)
ResponseEntity<Void> deleteMedicine(Long id)
ResponseEntity<MedicineCartDto> getMedicineCart(Long userId)
uses
Service Layer
IMedicineService
List<Medicine> getAllMedicines()
List<Medicine> searchMedicines(String name, String category)
Medicine saveMedicine(Medicine medicine)
void deleteMedicine(Long id)
Medicine updateMedicine(Long id, Medicine medicine)
MedicineCartDto getMedicineCart(Long userId)
implements
MedicineService
List<Medicine> getAllMedicines()
List<Medicine> searchMedicines(String name, String category)
Medicine saveMedicine(Medicine medicine)
void deleteMedicine(Long id)
Medicine updateMedicine(Long id, Medicine medicine)
MedicineCartDto getMedicineCart(Long userId)
returns
interacts
DTO Layer
MedicineCartDto
Long id
Long userId
Long cartId
contains
MedicineCartItemDto
Long id
Long userId
Long cartId
Long productId
Integer quantity
BigDecimal unitPrice
BigDecimal totalPrice
void setTotalPrice()
Repository Layer
MedicineRepository
List<Medicine> findByNameContainingIgnoreCase(String name)
List<Medicine> findByCategoryContainingIgnoreCase(String category)
boolean existsById(Long id)
manages
Model Layer
Medicine
Long id
String name
String description
String category
Double price
Integer stockQuantity

5. Hospital Stat(statistics)

The part(main functional requirements) is designed to help patients and hospital owners make informed decisions by providing insights into hospital performance across different regions. It achieves this by visualizing hospital statistics in a bar chart format, representing:

- **Country-wide hospital performance: Identifying the best hospitals across the country based on the total number of patients.**
- **Regional hospital performance: Highlighting the top-performing hospitals in specific areas, aiding patients in selecting the most suitable hospitals nearby.**
- **Branch performance insights: For hospital owners, this visualization helps determine which branches perform best, enabling better strategic planning and resource allocation.**

Features

- **Hospital Performance Analysis:**
  - **Patients can view the overall best hospitals.**
  - **Hospital owners can identify their top-performing branches.**
- **Area-wise Insights:**
  - **Allows patients to select the best hospital in their area.**
  - **Helps hospital owners understand which regions contribute significantly to their hospital's success.**
- **Branch Comparison for Hospitals:**
  - **A detailed analysis of patient distribution across**

  - hospital branches in different areas.
  - Aids in business strategy formulation for better service delivery.

Design Patterns Used

- **Service Layer Pattern:**

  - The service classes, `HospitalService` and `HospitalAppointmentService`, act as intermediaries between the repository and controller layers.
  - These classes encapsulate business logic, such as retrieving hospital statistics, adding appointments, and performing specific queries.
  - Benefits: Promotes separation of concerns, making the codebase more maintainable and testable.

- **Repository Pattern:**

  - The `HospitalRepository` and `HospitalAppointmentRepository` handle database operations using Spring Data JPA.
  - The repositories abstract the data access layer and allow for custom queries to fetch hospital statistics or area-specific insights.
  - Benefits: Decouples database access logic from business logic, simplifying code management and improving scalability.

- **Strategy Pattern (Query Customization):**

- Custom queries in the repositories, such as `findByNameLongestMatch` and `getHospitalStats`, are specific strategies to fetch data based on different criteria (e.g., name, area).
- Benefits: Each query acts as a tailored solution, supporting dynamic data retrieval based on business needs.

- Singleton Pattern:
  - Spring's `@Service` and `@Repository` annotations create single instances of these components within the application context.
  - Benefits: Ensures efficient resource usage and consistency across the application.

- Observer Pattern (Visualization):
  - The bar chart visualization acts as an observer, reflecting real-time changes in the data retrieved from the backend services.
  - Benefits: Patients and hospital owners can dynamically view updated statistics without manually refreshing or recalculating.

Technical Implementation

- Backend Implementation:
  - Service and repository layers handle complex queries for hospital statistics.
  - Custom queries such as `getHospitalStats`, `getAreaStatsByName`, and

`getHospitalStatsByArea` aggregate data for visualization.

Frontend Visualization:

- **The retrieved data is presented in bar charts, enabling easy comparison and analysis of hospital performance.**
- **Libraries such as Chart.js or D3.js may be used to render the graphical insights.**

Conclusion

The telemedicine web project successfully integrates diverse functionalities to provide an efficient and user-friendly platform for healthcare services. Leveraging robust software design patterns such as Builder, Strategy, Singleton, and Repository, the project ensures scalability, maintainability, and seamless interaction between components. The service layer effectively manages core operations, including appointment scheduling, user authentication, blood donor management, notifications, and e-commerce functionalities like cart and category management. By combining innovative technical solutions with a focus on user needs, the project enhances accessibility to healthcare while promoting transparency and efficiency. This comprehensive approach demonstrates the potential of technology to bridge gaps in traditional healthcare delivery systems, creating a scalable foundation for future enhancements.

Video Presentation:

sdp(Easy Care)