# **Software Design Specification (SDS)**

# **Hospital Management System**

## **1. Introduction**

### **1.1 Purpose**

This Software Design Specification (SDS) document presents the architectural and detailed design for the Hospital Management System (HMS). It serves as a blueprint for the implementation phase, detailing how the requirements specified in the SRS will be realized.

### **1.2 Scope**

This document covers the system architecture, component design, database design, interface design, and algorithm specifications for the Hospital Management System, including the three primary modules: Receptionist (Admin), Doctor, and Store Manager.

### **1.3 References**

* Software Requirements Specification for Hospital Management System
* Modern Web Application Design Patterns
* Database Design Best Practices for Healthcare Systems

### **1.4 Overview**

The remaining sections of this document provide detailed design information, starting with the architectural design, followed by the detailed component designs, user interface designs, database design, and implementation considerations.

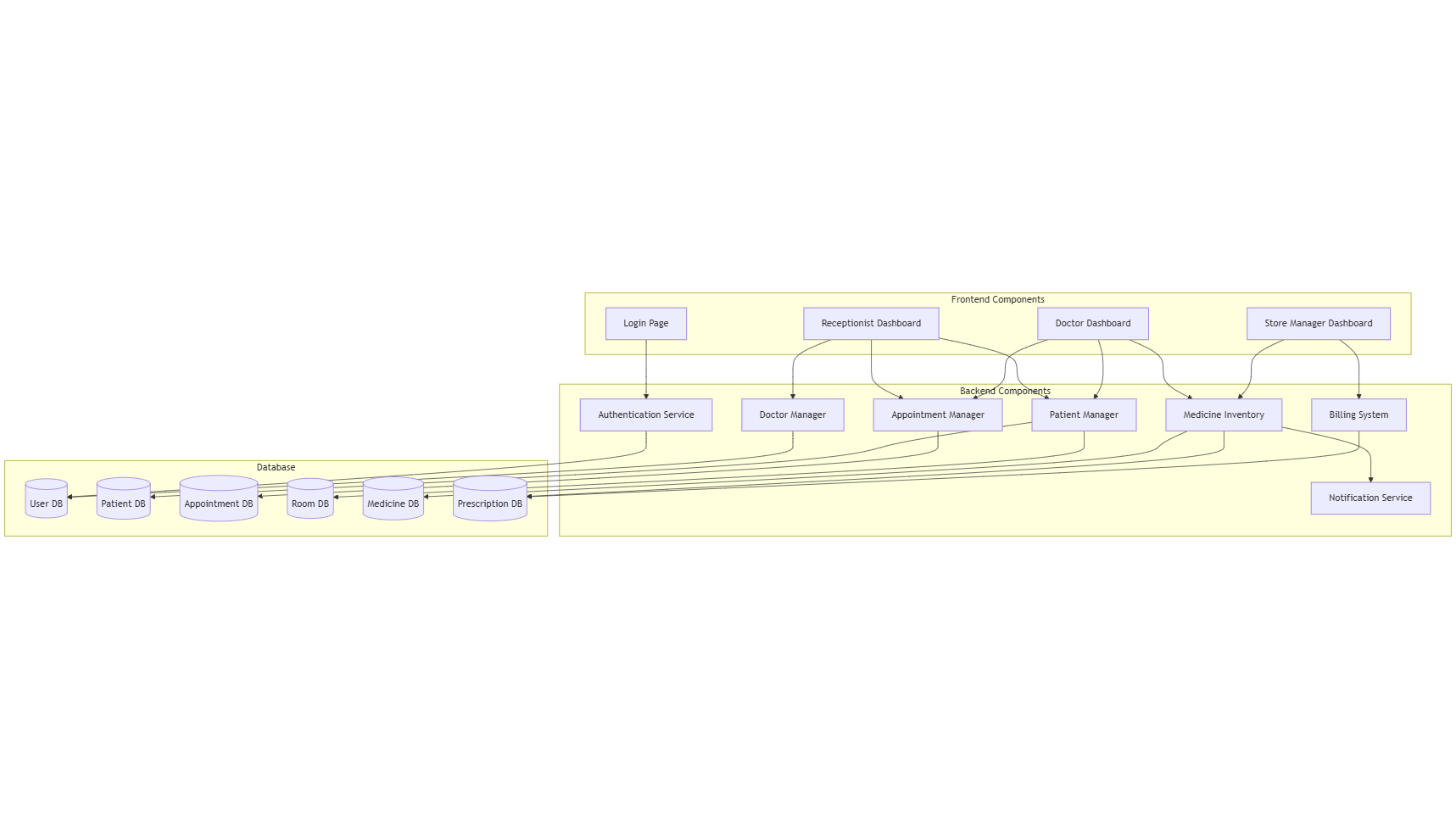
## **2. Architectural Design**

### **2.1 System Architecture Overview**

The Hospital Management System will be implemented using a three-tier architecture:

1. **Presentation Tier**: Contains the user interfaces for the three different user roles
2. **Application Tier**: Contains the business logic and application functionality
3. **Data Tier**: Manages data storage, retrieval, and persistence

**COMPONENT DIAGRAM:**



**2.2 Design Patterns and Principles**

The system will utilize the following design patterns and principles:

#### **2.2.1 MVC (Model-View-Controller) Pattern**

* **Model**: Represents the data structures and business logic
* **View**: Represents the user interface components
* **Controller**: Manages the flow of data between Model and View

#### **2.2.2 Singleton Pattern**

Used for components that should have only one instance, such as database connection managers.

#### **2.2.3 Factory Pattern**

Used for creating objects without specifying the exact class of object that will be created, particularly for creating user objects of different types.

#### **2.2.4 Repository Pattern**

Used for data access, abstracting the details of data persistence.

### **2.3 Security Architecture**

The security architecture includes:

* Authentication mechanism using email and password
* Role-based access control
* Session management
* Data encryption for sensitive information
* Input validation and sanitization

## **3. Component Design**

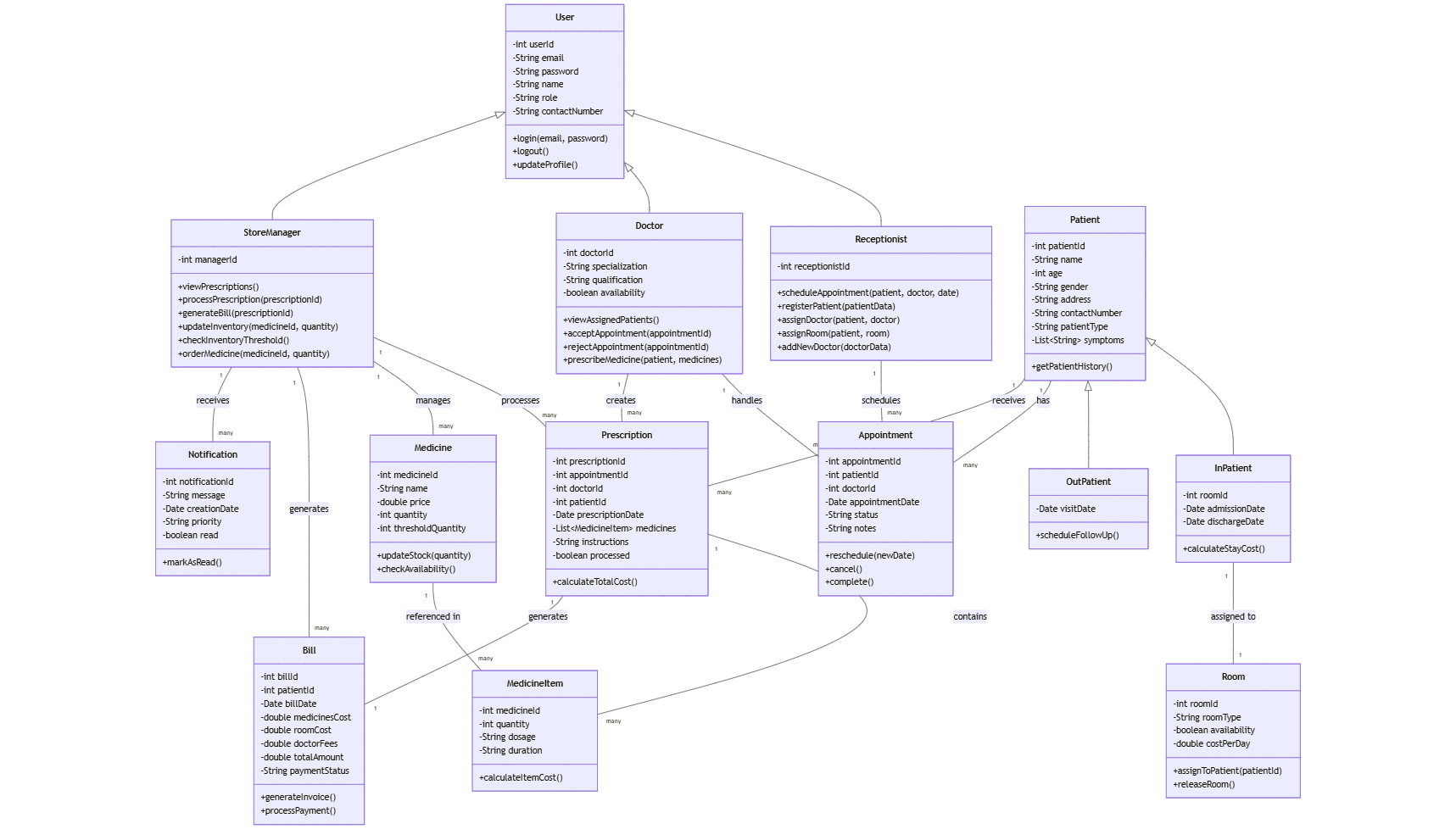
### **3.1 Authentication Component**

#### **3.1.1 Component Description**

This component handles user authentication, authorization, and session management.

#### **3.1.2 Class Structure**

**CLASS DIAGRAM PLACEMENT**

****

**3.1.3 Interface Definitions**

interface IAuthenticationService {

login(email: string, password: string): User | null;

logout(sessionId: string): boolean;

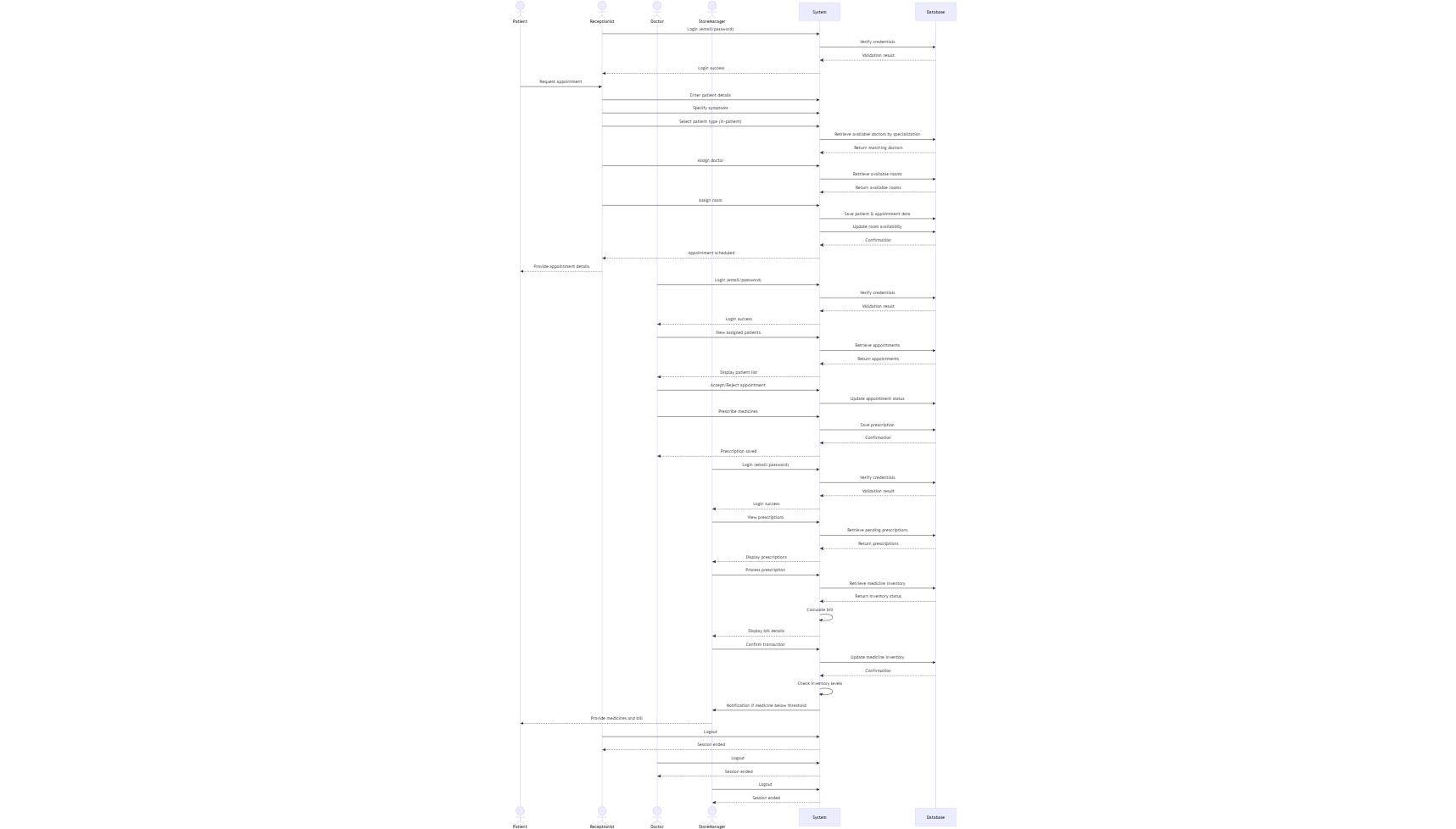
registerUser(userData: UserData): User | null;

validateSession(sessionId: string): boolean;

}

#### **3.1.4 Behavior Description**

**SEQUENCE DIAGRAM PLACEMENT**

****

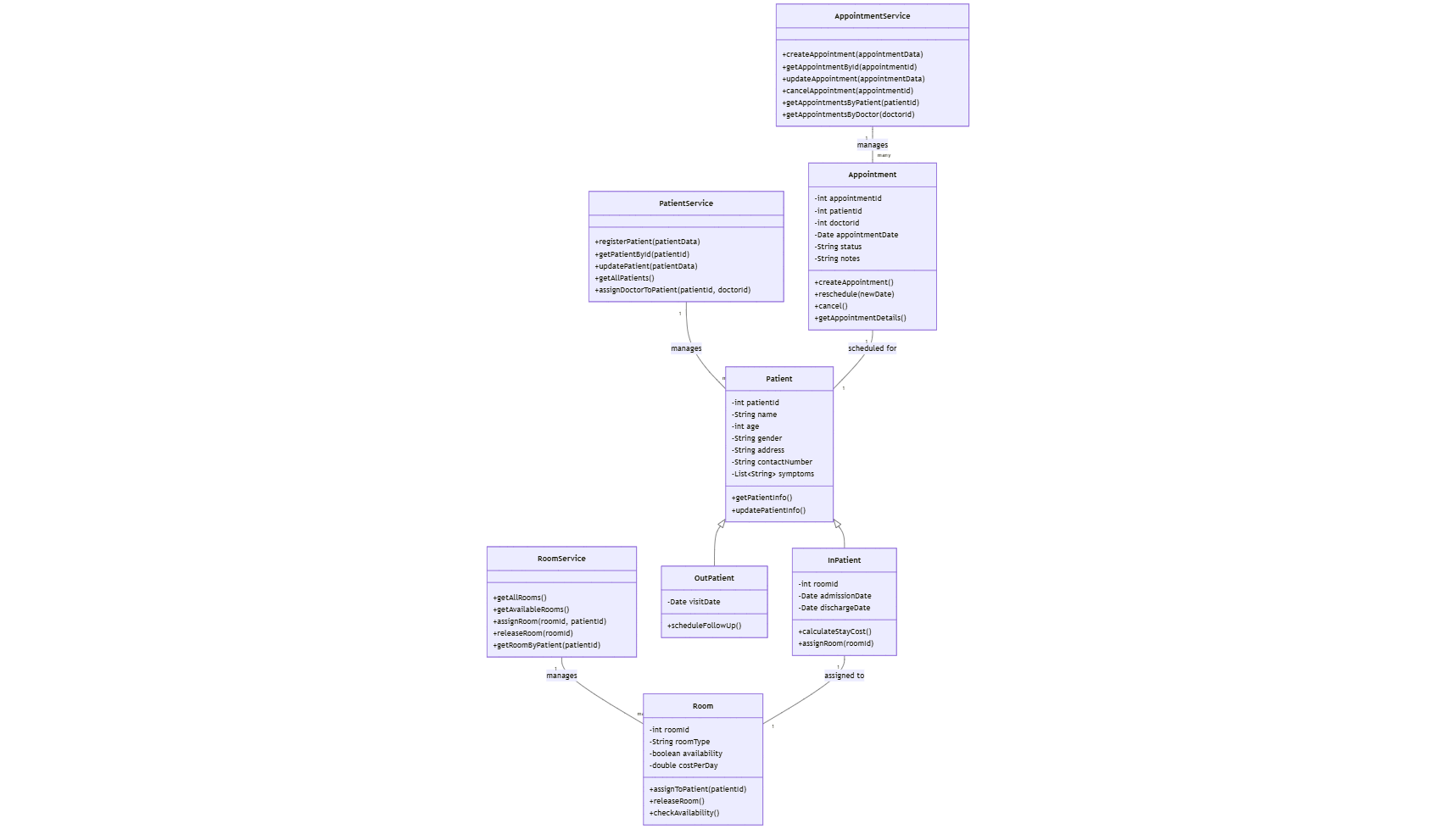
**3.2 Receptionist Component**

#### **3.2.1 Component Description**

This component provides functionality for patient registration, appointment scheduling, doctor assignment, and room allocation.

#### **3.2.2 Class Structure**

**CLASS DIAGRAM:**

****

**3.2.3 Interface Definitions**

interface IPatientService {

registerPatient(patientData: PatientData): Patient | null;

getPatientById(patientId: string): Patient | null;

updatePatient(patientId: string, patientData: PatientData): boolean;

}

interface IAppointmentService {

createAppointment(appointmentData: AppointmentData): Appointment | null;

getAppointmentsByDoctor(doctorId: string): Appointment[];

getAppointmentsByPatient(patientId: string): Appointment[];

updateAppointmentStatus(appointmentId: string, status: AppointmentStatus): boolean;

}

interface IRoomService {

getAllAvailableRooms(): Room[];

assignRoomToPatient(roomId: string, patientId: string): boolean;

releaseRoom(roomId: string): boolean;

}

#### **3.2.4 Behavior Description**

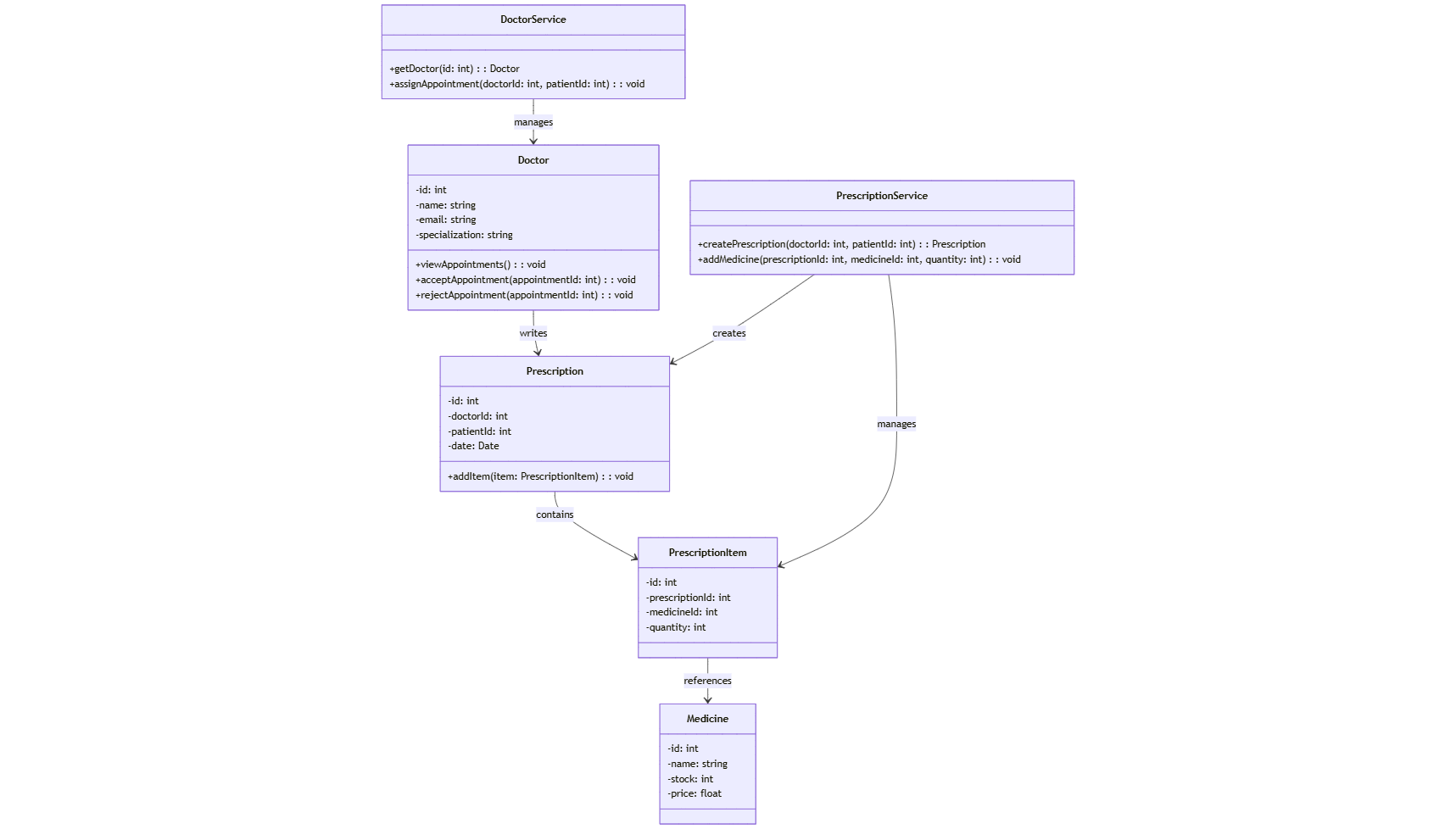
### 

### **3.3 Doctor Component**

#### **3.3.1 Component Description**

This component provides functionality for doctors to manage appointments and create prescriptions.

#### **3.3.2 Class Structure**

[**CLASS DIAGRAM PLACEMENT**

#### **3.3.3 Interface Definitions**

interface IDoctorService {

getAssignedPatients(doctorId: string): Patient[];

getAppointments(doctorId: string, status?: AppointmentStatus): Appointment[];

updateAppointmentStatus(appointmentId: string, status: AppointmentStatus): boolean;

}

interface IPrescriptionService {

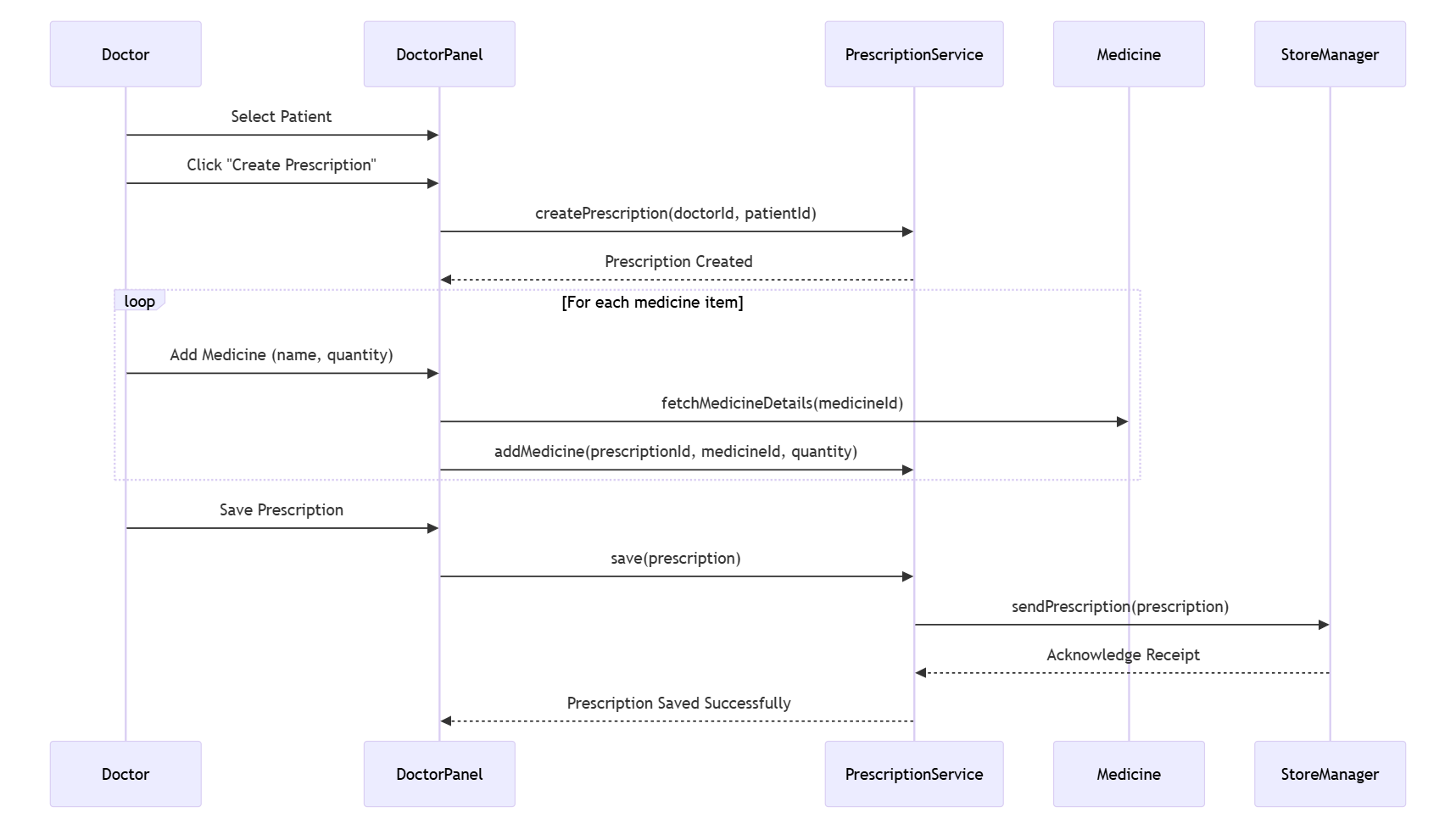
createPrescription(prescriptionData: PrescriptionData): Prescription | null;

addItemToPrescription(prescriptionId: string, itemData: PrescriptionItemData): boolean;

getPrescriptionsByPatient(patientId: string): Prescription[];

}

#### **3.3.4 Behavior Description**

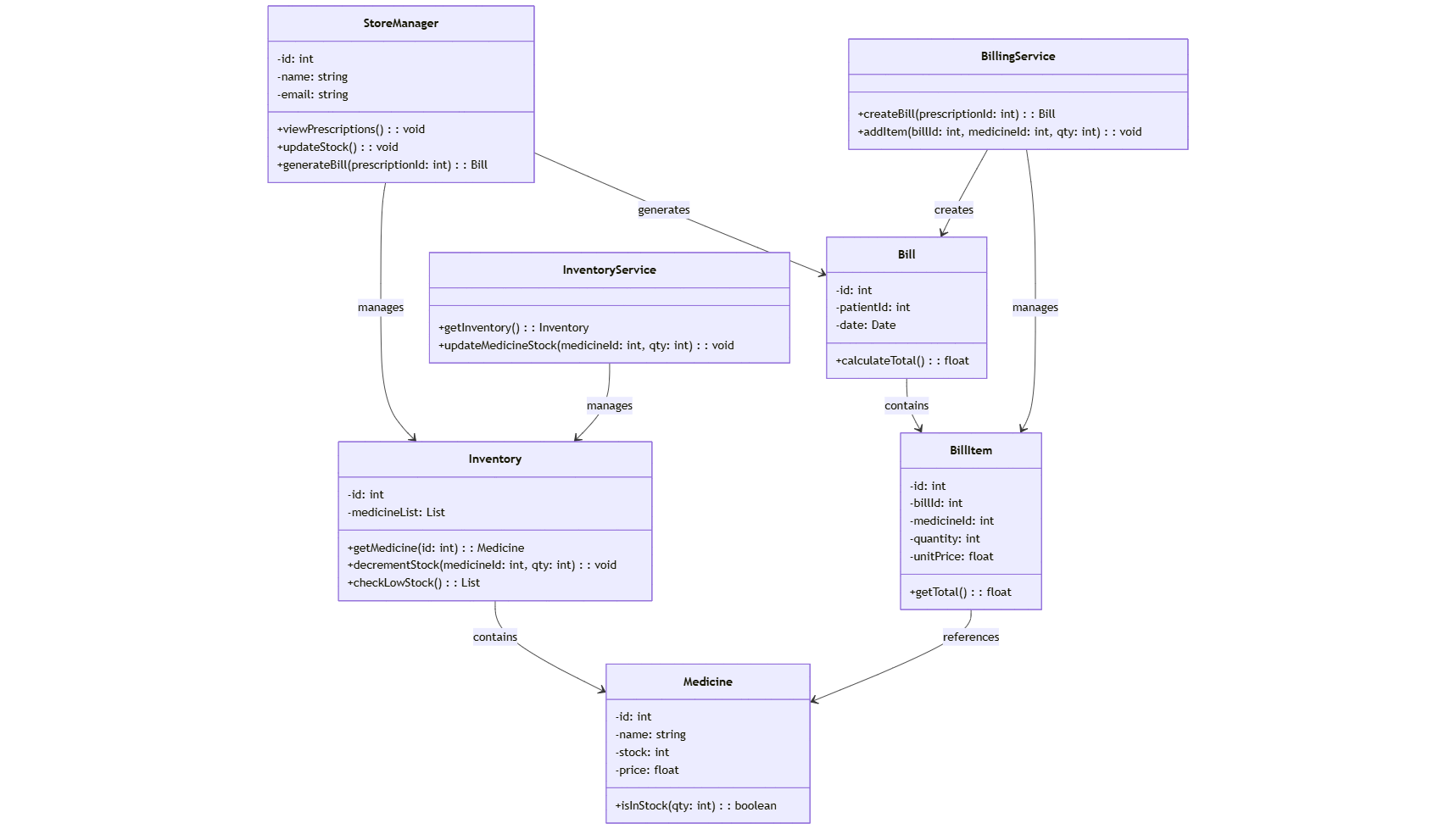
* 

### **3.4 Store Manager Component**

#### **3.4.1 Component Description**

This component provides functionality for managing medicine inventory, processing prescriptions, and generating bills.

#### **3.4.2 Class Structure**

* 

#### **3.4.3 Interface Definitions**

interface IInventoryService {

getMedicineStock(medicineId: string): number;

updateStock(medicineId: string, change: number): boolean;

checkLowStock(): Medicine[];

addNewMedicine(medicineData: MedicineData): Medicine | null;

}

interface IBillingService {

generateBill(prescriptionId: string): Bill | null;

calculateTotalAmount(billId: string): number;

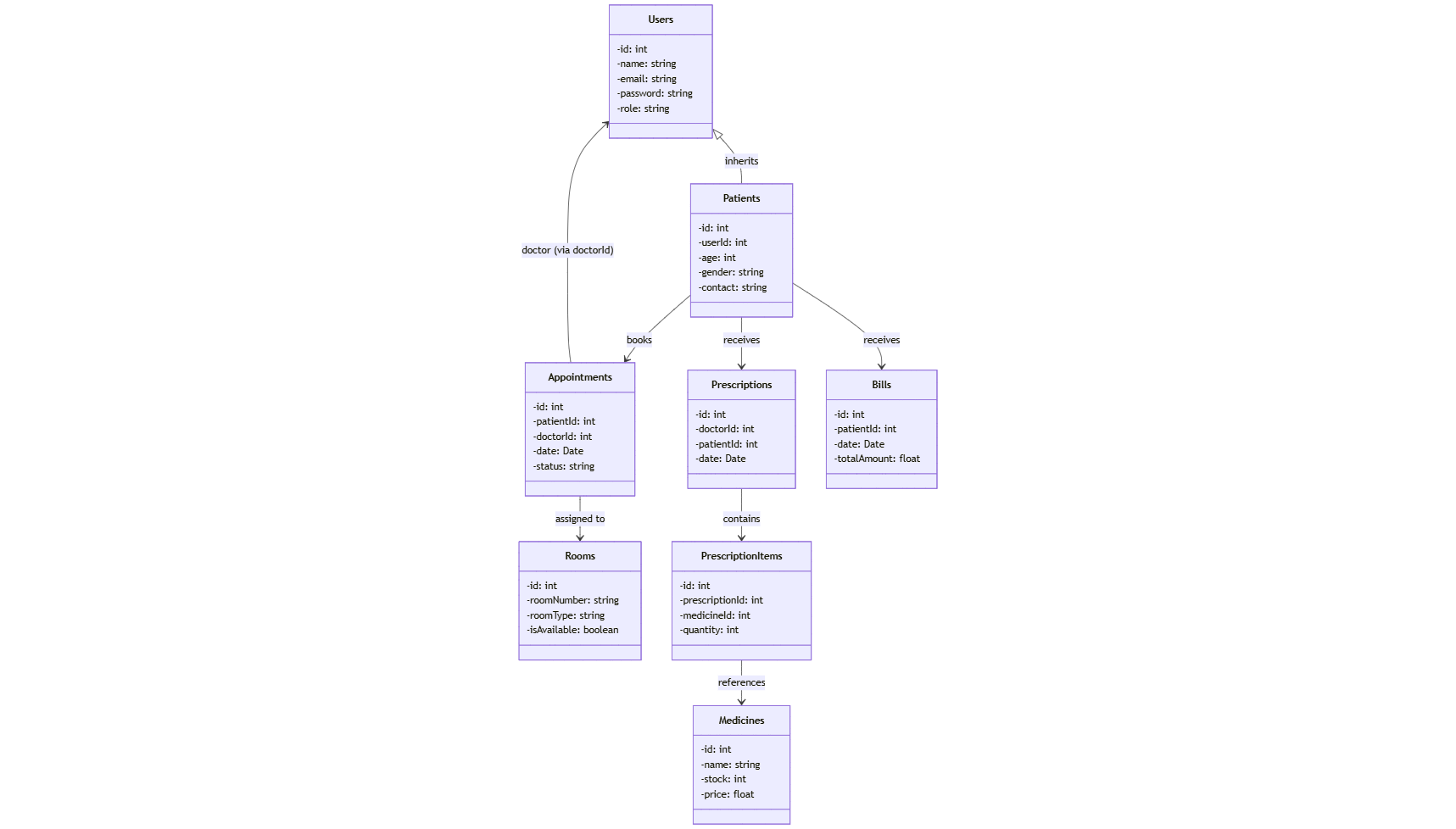
getBillsByPatient(patientId: string): Bill[];

}

## **4. Data Design**

### **4.1 Database Schema**

#### **4.1.1 Entity-Relationship Diagram**

* 

#### **4.1.2 Table Definitions**

##### **Users Table**

* UserID (PK)
* Name
* Email
* PasswordHash
* Role
* ContactNumber
* CreatedAt
* LastLogin

##### **Patients Table**

* PatientID (PK)
* Name
* Age
* Gender
* ContactNumber
* Address
* PatientType (In/Out)
* AssignedDoctorID (FK)
* RoomID (FK, Nullable)
* RegistrationDate

##### **Appointments Table**

* AppointmentID (PK)
* PatientID (FK)
* DoctorID (FK)
* AppointmentDate
* AppointmentTime
* Status
* Notes

##### **Rooms Table**

* RoomID (PK)
* RoomNumber
* RoomType
* Status
* PricePerDay

##### **Prescriptions Table**

* PrescriptionID (PK)
* PatientID (FK)
* DoctorID (FK)
* PrescriptionDate
* Status

##### **PrescriptionItems Table**

* ItemID (PK)
* PrescriptionID (FK)
* MedicineID (FK)
* Quantity
* Dosage
* Instructions

##### **Medicines Table**

* MedicineID (PK)
* Name
* Description
* CurrentStock
* MinimumStockThreshold
* PricePerUnit

##### **Bills Table**

* BillID (PK)
* PatientID (FK)
* PrescriptionID (FK)
* BillDate
* TotalAmount
* Status

### **4.2 Data Access Layer**

The data access layer will use the Repository pattern to abstract database operations. Each entity will have a corresponding repository class responsible for CRUD operations.

## **5. User Interface Design**

### **5.1 General Interface Design Principles**

* Clean and intuitive layout
* Consistent color scheme and typography
* Responsive design for different screen sizes
* Clear navigation between different sections
* Feedback for user actions
* Accessibility considerations

### **5.2 Receptionist Interface Design**

#### **5.2.1 Login Screen**

* Email and password fields
* Login button
* Forgot password link

#### **5.2.2 Dashboard**

* Summary of pending appointments
* Quick access to patient registration
* Doctor availability status
* Room availability status

#### **5.2.3 Patient Registration Form**

* Personal information section
* Patient type selection (In/Out)
* Symptoms and notes section
* Doctor selection dropdown
* Room selection (for In-patients)
* Submit button

#### **5.2.4 Appointment Management**

* Calendar view of appointments
* Filter options by date, doctor, status
* Appointment details view
* Rescheduling functionality

### **5.3 Doctor Interface Design**

#### **5.3.1 Login Screen**

* Email and password fields
* Login button
* Forgot password link

#### **5.3.2 Dashboard**

* Upcoming appointments
* Patient count
* Recent activities

#### **5.3.3 Appointment Management**

* List of appointments
* Accept/Reject buttons
* Patient details view

#### **5.3.4 Prescription Creation**

* Patient selection
* Medicine search and selection
* Dosage and instructions fields
* Add button for multiple medicines
* Submit button

### **5.4 Store Manager Interface Design**

#### **5.4.1 Login Screen**

* Email and password fields
* Login button
* Forgot password link

#### **5.4.2 Dashboard**

* Pending prescriptions count
* Low stock alerts
* Recent billing activities

#### **5.4.3 Prescription Processing**

* List of prescriptions
* Prescription details view
* Process button
* Generate bill button

#### **5.4.4 Inventory Management**

* Medicine list with stock levels
* Add new medicine form
* Update stock form
* Stock history view

## **6. Algorithm Specifications**

### **6.1 Doctor Assignment Algorithm**

function assignDoctorToPatient(patientSymptoms, availableDoctors):

matchScore = {}

for each doctor in availableDoctors:

score = calculateMatchScore(patientSymptoms, doctor.specialization)

matchScore[doctor.id] = score

bestMatch = findDoctorWithHighestScore(matchScore)

if bestMatch.score > threshold:

return bestMatch.doctor

else:

return getGeneralPhysician(availableDoctors)

### **6.2 Room Allocation Algorithm**

function allocateRoomToPatient(patientType, patientCondition):

if patientType != "In-patient":

return null

availableRooms = getAvailableRooms()

if patientCondition == "Critical":

room = findAvailableRoomByType(availableRooms, "ICU")

else if patientCondition == "Serious":

room = findAvailableRoomByType(availableRooms, "Special")

else:

room = findAvailableRoomByType(availableRooms, "General")

if room:

markRoomAsOccupied(room.id)

return room

### **6.3 Low Stock Alert Algorithm**

function checkForLowStock():

allMedicines = getAllMedicines()

lowStockItems = []

for each medicine in allMedicines:

if medicine.currentStock <= medicine.minimumStockThreshold:

lowStockItems.add(medicine)

if lowStockItems.length > 0:

notifyStoreManager(lowStockItems)

return lowStockItems

## **7. Implementation Considerations**

### **7.1 Development Tools and Environment**

* **Frontend**: HTML5, CSS3, JavaScript, React.js/Angular/Vue.js
* **Backend**: Node.js with Express/Java with Spring Boot/Python with Django
* **Database**: MongoDB/MySQL/PostgreSQL
* **Version Control**: Git with GitHub/GitLab
* **IDE**: Visual Studio Code/IntelliJ IDEA/Eclipse
* **Testing Tools**: Jest, Mocha, JUnit, Selenium

### **7.2 Coding Standards and Guidelines**

* Follow consistent naming conventions
* Use proper code indentation and formatting
* Include comments for complex logic
* Create comprehensive documentation
* Write unit tests for all components
* Use linting tools for code quality

### **7.3 Error Handling and Logging**

* Implement structured error handling
* Use try-catch blocks for exception handling
* Create meaningful error messages
* Implement logging at different levels (info, warning, error)
* Log important events and errors
* Implement monitoring for system issues

### **7.4 Performance Considerations**

* Optimize database queries
* Implement caching where appropriate
* Use pagination for large data sets
* Minimize network requests
* Optimize front-end assets
* Consider load balancing for high traffic

## **8. Testing Strategy**

### **8.1 Unit Testing**

Each component and class will be tested in isolation to ensure they function correctly.

### **8.2 Integration Testing**

Components will be tested together to ensure they work correctly when integrated.

### **8.3 System Testing**

The entire system will be tested to ensure it meets the requirements specified in the SRS.

### **8.4 User Acceptance Testing**

End users will test the system to ensure it meets their expectations and requirements.

### **8.5 Test Cases (Sample)**

#### **8.5.1 Authentication Test Cases**

* TC001: User login with valid credentials
* TC002: User login with invalid credentials
* TC003: User logout
* TC004: Password reset

#### **8.5.2 Receptionist Module Test Cases**

* TC101: Register new in-patient
* TC102: Register new out-patient
* TC103: Schedule appointment
* TC104: Assign doctor to patient
* TC105: Allocate room to in-patient

#### **8.5.3 Doctor Module Test Cases**

* TC201: View assigned patients
* TC202: Accept appointment
* TC203: Reject appointment
* TC204: Create prescription

#### **8.5.4 Store Manager Module Test Cases**

* TC301: Process prescription
* TC302: Generate bill
* TC303: Update medicine stock
* TC304: Receive low stock notification

## **9. Deployment Considerations**

### **9.1 Deployment Architecture**

* Web server configuration
* Database server setup
* Application server setup
* Load balancer (if required)
* Backup and recovery systems

### **9.2 Installation and Setup**

* Database initialization
* Application installation
* Configuration settings
* Initial data setup
* User account creation

### **9.3 Maintenance and Support**

* Regular backups
* System monitoring
* Performance tuning
* Security updates
* Bug fixes and patches

## **10. Appendices**

### **10.1 Glossary**

* **HMS**: Hospital Management System
* **UI**: User Interface
* **DB**: Database
* **API**: Application Programming Interface
* **CRUD**: Create, Read, Update, Delete

### **10.2 References**

* Software Requirements Specification for Hospital Management System
* Modern Web Application Architecture Best Practices
* Healthcare Application Security Guidelines

### **10.3 Sample API Documentation**

#### **10.3.1 Authentication API**

**POST /api/auth/login**

* Request Body: { email, password }
* Response: { token, user }

**POST /api/auth/logout**

* Request Header: Authorization: Bearer {token}
* Response: { success: true }

#### **10.3.2 Patient API**

**POST /api/patients**

* Request Header: Authorization: Bearer {token}
* Request Body: { name, age, gender, contactNumber, patientType, symptoms }
* Response: { patientId, name, ... }

**GET /api/patients/:id**

* Request Header: Authorization: Bearer {token}
* Response: { patientId, name, age, ... }

#### **10.3.3 Appointment API**

**POST /api/appointments**

* Request Header: Authorization: Bearer {token}
* Request Body: { patientId, doctorId, date, time }
* Response: { appointmentId, patientId, ... }

**PUT /api/appointments/:id/status**

* Request Header: Authorization: Bearer {token}
* Request Body: { status }
* Response: { appointmentId, status, ... }

#### **10.3.4 Prescription API**

**POST /api/prescriptions**

* Request Header: Authorization: Bearer {token}
* Request Body: { patientId, doctorId, items: [{ medicineId, quantity, dosage, instructions }] }
* Response: { prescriptionId, patientId, ... }

**GET /api/prescriptions/:id**

* Request Header: Authorization: Bearer {token}
* Response: { prescriptionId, patientId, items: [...] }